Formale Semantik des Datentypmodells von SDL-2000

D i s s e r t a t i o n

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (doctor rerum naturalium)

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von
Dipl-Inf. Martin von Löwis of Menar
geboren am 25.4.1970 in Berlin

Präsident / Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr. Jürgen Mlynek

_____

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Elmar Kulke

_____

Gutachter / Gutachterin

1. Prof. Dr. Joachim Fischer

2. Prof. Dr. Reinhard Gotzhein

3. Prof. Dr. Andreas Mitschele-Thiel

Tag der mündlichen Prüfung 18.12.2003

Zusammenfassung

Mit der aktuellen Überarbeitung der Sprache SDL (Specification and Description Language) der ITU-T wurde die semantische Fundierung der formalen Definition dieser Sprache vollständig überarbeitet; die formale Definition basiert nun auf dem Kalkül der Abstract State Machines (ASMs). Ebenfalls neu definiert wurde das um objekt-orientierte Konzepte erweiterte Datentypsystem. Damit musste eine formale semantische Fundierung für diese neuen Konzepte gefunden werden. Der bisher verwendete Kalkül ACT.ONE sollte nicht mehr verwendet werden, da er schwer verwendbar, nicht implementierbar und nicht auf Objektsysteme erweiterbar ist.

In der vorliegenden Arbeit werden die Prinzipien einer formalen Sprachdefinition dargelegt und die Umsetzung dieser Prinzipien für die Sprache SDL-2000 vorgestellt. Dabei wird erläutert, dass eine konsistente Sprachdefinition nur dadurch erreicht werden konnte, dass die Definition der formalen Semantik der Sprache parallel mit der Entwicklung der informalen Definition erfolgte. Dabei deckt die formale Sprachdefinition alle Aspekte der Sprache ab: Syntax, statische Semantik und dynamische Semantik. Am Beispiel der Datentypsemantik wird erläutert, wie jeder dieser Aspekte informal beschrieben und dann formalisiert wurde.

Von zentraler Rolle für die Anwendbarkeit der formalen Semantikdefinition in der Praxis ist der Einsatz von Werkzeugen. Die Arbeit erläutert, wie aus der formalen Sprachdefinition vollautomatisch ein Werkzeug generiert wurde, das die Sprache SDL implementiert, und wie die durch die Umsetzung der formalen Semantikdefinition in ein Werkzeug Fehler in dieser Definition aufgedeckt und behoben werden konnten.

Abstract

With the latest revision of ITU-T SDL (Specification and Description Language), the semantic foundations of the formal language definition were completely revised; the formal definition is now based on the calculus of Abstract State Machines (ASMs). In addition, the data type system of SDL was revised, as object-oriented concepts were added. As a result, a new semantical foundation for these new concepts had to be defined. The ACT.ONE calculus that had been used so far was not suitable as a foundation any more, as it is hard to use, unimplementable and not extensible for the object oriented features.

In this thesis, we elaborate the principles of a formal language definition, and the realisation of these principles in SDL-2000. We explains that a consistent language definition can only be achieved by developing the formal semantics definition in parallel with the development of the informal definition. The formal language definition covers all aspects of the language: syntax, static semantics, and dynamic semantics. Using the data type semantics as an example, we show how each of these aspects is informally described, and then formalized.

For the applicability of the formal semantics definition for practitioners, usage of tools plays a central role. We explain how we transform the formal language definition fully automatically into a tool that implements the language SDL. We also explain how creating the tool allowed us to uncover and correct errors in the informal definition.

Inhaltsverzeichnis

1 Einleitung

Mit der Entwicklung von Sprachen und Notationen, die vom Menschen benutzt und vom Computer verstanden werden sollen, entstand der Wunsch, solche Sprachen (im folgenden Computersprachen genannt) möglichst unzweideutig zu beschreiben.

Dazu wurde versucht, die Beschreibung der jeweiligen Computersprache durch eine natürliche Sprache (in der Regel Englisch) möglichst „wasserdicht“ zu machen, also Formulierungen zu wählen, die eine Fehlinterpretation (absichtlich oder unabsichtlich) erschweren. In der Praxis ist dies nicht immer gelungen: Die Beschreibung war manchmal widersprüchlich, unvollständig, oder falsch. Eine Beschreibung wird *widersprüchlich* genannt, wenn zwei Forderungen an ein Programm gestellt werden, die nicht gleichzeitig erfüllt sein können. Sie ist *unvollständig*, wenn für ein bestimmtes Programm in wesentlichen Punkten keine Aussage getroffen wird, beispielsweise, ob es richtig oder falsch ist, oder wie es sich unter bestimmten Umständen verhält. Die Beschreibung ist falsch, wenn sie nicht ausdrückt, was ihre Autoren eigentlich ausdrücken wollten.

Widersprüchliche oder unvollständige Beschreibungen kann ein Leser (einer solchen Sprachbeschreibung) selbst erkennen: Ist die Beschreibung widersprüchlich, so gibt es Programme, die nach der Beschreibung Eigenschaften haben, die sie gleichzeitig nicht haben können, weil sich diese Eigenschaften ausschließen. Ist die Beschreibung unvollständig, so gibt es Programme, die mehrere alternative Eigenschaften haben könnten, und die Sprachdefinition sagt nicht, welche der Alternativen für diese Sprache verwendet werden soll.

Ist die Sprachdefinition falsch, kann das ein Leser nicht unbedingt entdecken: Die Definition entspricht dann nicht den Intentionen der Autoren der Definition. Um derartige Fehler zu erkennen, muss der Leser die Intentionen der Autoren kennen (beispielsweise selbst Autor der Sprachdefinition sein).

Um klare Sprachbeschreibungen zu erhalten, wurde versucht, zusätzlich zu der informalen Beschreibung formale Sprachdefinitionen zu schaffen. Diese wurden nicht in einer natürlichen Sprache definiert, sondern benutzen eine mathematische Theorie und mathematische Notationen, um die Sprache zu definieren.

Mit einer formalen Sprachdefinition verschwinden einige der genannten Probleme: eine formale Definition kann so konstruiert werden, dass sie vollständig und widerspruchsfrei ist. Dazu muss die mathematische Theorie so ausgewählt werden, dass sie in jedem Fall, also für jedes betrachtete Programm der Sprache stets genau eine Interpretation des Programms liefert – oder aber eine Menge möglicher Interpretationen, die dann alle als konforme Interpretationen betrachtet werden.

Dennoch kann man mit einer formalen Sprachdefinition nicht sicherstellen, dass sie richtig ist, da dazu wiederum die Intentionen der Autoren formalisiert sein müssten. Man kann aber die Aufdeckung von Fehlern erleichtern: Falls sowohl die informale Definition als auch die formale Definition eine Interpretation eines Programms liefern und die Interpretationen voneinander abweichen, ist mit Sicherheit eine der beiden Sprachdefinitionen falsch. Welche der Definitionen dann die richtige ist, können nur die Autoren der Sprachdefinitionen beantworten.

1.1 Ziele der Arbeit

In dieser Arbeit wird die formale Definition der Datentypsemantik von SDL-2000 vorgestellt. SDL-2000 (im Folgenden auch einfach SDL genannt) ist die aktuelle Version der Sprache ITU-T Specification and Description Language [Z.100-00]. SDL wird in der Telekommunikationsindustrie für die Beschreibung reaktiver Systeme eingesetzt.

Während der Entwicklung von SDL-2000 wurde klar, dass die formale Semantikdefinition, die für die Vorgängerversion SDL-92 veröffentlicht wurde, nicht auf SDL-2000 angepasst wer-

den kann. Verschiedene Forschungsgruppen waren unabhängig voneinander zu dem Ergebnis gekommen, dass die Vielzahl der in der Vorgängerversion verwendeten Kalküle ein genaues Verständnis der formalen Sprachdefinition unmöglich macht, und hatten versucht, die SDL-Semantik auf der Basis konsistenter Kalküle zu definieren, darunter:

- die Definition auf Basis von stream processing functions in [Broy91],
- die Definition einer Kernsprache BSDL in [FLP95] und [LP95] sowie
- die Definition der SDL-Semantik auf Basis eines Transitionssystems in [GGRS98].

Diese Arbeitsgruppen kamen schließlich gemeinsam zu der Überzeugung, dass eine Neudefinition der formalen Semantik von SDL auf der Basis eines einheitlichen Kalküls erforderlich ist. Deshalb wurde 1998 eine Projektgruppe etabliert, die die formale Semantik von SDL auf der Basis des Kalküls der Abstract State Machines definieren sollte. Diese Projektgruppe bestand aus folgenden Mitgliedern:

- Bo Ai, Beijing University of Posts & Telecommunications (statische Semantik),
- Robert Eschbach, Universität Kaiserslautern (Dynamic Semantik),
- Uwe Glässer, Microsoft Research (Dynamische Semantik),
- Reinhard Gotzhein, Universität Kaiserslautern (ITU Study Group 10 Associate Rapporteur, dynamic semantics),
- Martin von Löwis of Menar, Humboldt University Berlin (Daten-Semantik),
- Andreas Prinz, DResearch, Berlin (Projectmanagement, Statische und dynamische Semantik),
- Ying Wang, Beijing University of Posts & Telecommunications (Statische Semantik),
- Weilei Zhang, Beijing University of Posts & Telecommunications (Statische Semantik) und
- Yuhong Zhao, Beijing University of Posts & Telecommunications (Statische Semantik).

Diese Gruppe hat zusammen mit den anderen SDL-Experten der ITU-Studiengruppe 10 die formale Semantik von SDL-2000 parallel zur Definition dieser Sprache entwickelt. Der Autor dieser Arbeit war unter anderem[1] an der informalen Definition des neuen Datentypteils von SDL-2000 beteiligt und hat die formale Definition dieses Teils entwickelt.

Der Datentypteil von SDL selbst hat in der Sprachversion SDL-2000 eine vollständige Überarbeitung erfahren. Auslöser für diese Änderungen waren Probleme beim Einsatz des Datentypteils von SDL-92 in praktischen Anwendungen, wie sie Ralf Schröder in [Sch94] und [Sch02] dargestellt hat.

Die vorliegende Arbeit stellt die Probleme dar, die sich bei der Definition der Datentypsemantik gestellt haben, und zeigt, wie diese Probleme gelöst wurden. Sie zeigt auch die Rahmenbedingungen, an die sich die formale Datentypsemantik anpassen musste:

- Die grundlegenden Kalküle sollten einheitlich für die gesamte Semantikdefinition sein, also musste sich die Datentypsemantik ebenfalls dieser Kalküle bedienen.
- Die informale Definition war ebenfalls vorgegeben, zumindest in ihren Konzepten.
- Die Datentypsemantik sollte einer funktionalen Schnittstelle (Abschnitt 8.1) unterliegen, um das Datentypsystem durch ein anderes ersetzen zu können, falls dies in der Praxis erforderlich würde.
- Die Datentypsemantik sollte, wie der Rest der formalen Definition, ausführbar sein.

Die Arbeit erläutert nun, wie diese Rahmenbedingungen eingehalten und eine verständliche Semantikdefinition des Datentypteils geschaffen wurde. Die Ergebnisse der Arbeit finden sich auf drei Gebieten: im Entwicklungsprozess einer formalen Sprachdefinition, in der Definition der Semantik selbst, und in den Werkzeugen, die zur Entwicklung eingesetzt wurden.

1. Der Autor hat auch an anderen Sprachkonzepten der SDL-2000-Definition mitgewirkt, etwa bei der Definition von Ausnahmen und Ausnahmebehandlung, dem Schnittstellekonzept sowie bei der Definition strukturierter Zustände.

Mit der Definition der formalen Semantik von SDL im Allgemeinen und der hier vorgestellten Datentypsemantik im Besonderen wird offenbar, dass die Entwicklung einer formalen Sprachdefinition zusammen mit der Weiterentwicklung der Sprache durchaus möglich ist. Es wurden aber auch etliche Probleme sichtbar, die mit dieser Arbeit dokumentiert werden.

Es hat sich insbesondere herausgestellt, dass mit der Formulierung der formalen Sprachdefinition allein diese Sprachdefinition nicht nutzbar ist, weil sie zum einen voll von Fehlern ist und zum anderen der Umfang des Formelwerks, der zur exakten Beschreibung der Semantik nötig ist, die Handhabung erschwert.

Diese Arbeit demonstriert, dass erst durch den Einsatz von Werkzeugen im Entwicklungsprozess der formalen Semantikdefinition diese praktisch anwendbar wird.

1.2 Publikum

Diese Arbeit richtet sich zum einen an Autoren von Sprachdefinitionen und -standards, die für „ihre“ Sprache eine formale Definition schaffen wollen oder bereits geschaffen haben und nun für diese formale Definition Werkzeuge entwickeln wollen. Zum anderen richtet sich die Arbeit an Forscher auf dem Gebiet formaler Methoden, die ermutigt werden sollen, die formale Semantik von SDL in ihre Arbeit einzubeziehen.

Anwender von SDL können dieser Arbeit die Funktionsweise der formalen SDL-Semantik entnehmen, um spezielle Aspekte der Semantik mit Hilfe der Werkzeuge genauer zu studieren.

In den Abhandlungen der Arbeit werden Grundbegriffe formaler Methoden wie beispielsweise Kenntnisse des Prädikatenkalküls vorausgesetzt. Zum Verständnis hilfreich ist weiterhin die Kenntnis einer Vorgängerversion von SDL-2000, da im Text nur die unmittelbar verwendeten SDL-Konstrukte erläutert werden, aber keine vollständige Darstellung der Sprache SDL zu finden ist.

1.3 Grundlagen der Arbeit

Die Arbeit basiert dabei auf drei Grundlagen: dem Compilerbau, der informalen Sprachdefinition von SDL, sowie dem Forschungsgebiet formaler Methoden. Die (informale) Sprachdefinition von SDL ist als [Z.100-00] veröffentlicht. Die Habilitation von Andreas Prinz zur Methodik einer formalen Sprachdefinition [Pri99] und die Diplomarbeit von Michael Piefel [Pie00] zu Teilen der verwendeten Werkzeugkette bilden Grundlagen auf dem Gebiet des Compilerbaus. Auf dem Gebiet der formalen Methoden basiert die Arbeit vor allem auf dem Kalkül der *Abstract State Machines* [Gur95] und der Entwicklung des Werkzeugs AsmL [Mic02].

1.4 Struktur der Arbeit

Jede der Grundlagen wird in einem eigenständigen Kapitel dargestellt: Kapitel 2 dieser Arbeit erläutert Techniken, die in der Vergangenheit für die formale Definition von Sprachen eingesetzt wurden. Kapitel 3 stellt die Sprache SDL vor. Kapitel 5 gibt einen Überblick über die formale Definition von SDL.

Auch die Ergebnisse werden in separaten Kapiteln vorgestellt: Kapitel 4 erläutert den Entwicklungsprozess der formalen Sprachdefinition aus Sicht des Autors. Zunächst kann die Entwicklung bei Einsatz eines geeigneten Satzes an formalen Techniken relativ systematisch erfolgen, indem die informale Sprachdefinition möglichst Wort für Wort formalisiert wird. Von dieser Systematik muss in dem Moment abgewichen werden, wo Probleme mit dieser Arbeitsweise auftauchen. Diese Probleme müssen dann analysiert und behoben werden. Kapitel 4 systematisiert die tatsächlich aufgefundenen Probleme und zeigt die möglichen Auswirkungen jedes Problems und seiner Behebung.

Die eigentliche Definition der Datentypsemantik von SDL findet der Leser in den Kapiteln 7 und 8. Kapitel 7 stellt die Definition der statischen Semantik, Kapitel 8 die der dynamischen Semantik des Datentypteils vor. Dabei wird keineswegs eine vollständige Vorstellung der Formalisierung angestrebt. Vielmehr werden für jede der eingesetzten Techniken typische Beispiele demonstriert. Um trotzdem einen Überblick über die gesamte Datentypsemantik zu ermöglichen, wird in Kapitel 6 ein Teil der formalen Datentypsemantik (die abstrakte Syntax 0) vollständig wiedergegeben und mit Erläuterungen versehen.

Kapitel 9 schließlich erläutert die im Rahmen dieser Arbeit entstandenen Werkzeuge zur Verarbeitung der formalen Sprachdefinition. Ohne die Werkzeuge wären Aussagen zur Richtigkeit und Vollständigkeit der formalen Sprachdefinition nicht glaubhaft zu machen, da erst die Werkzeuge systematische Konsistenzprüfungen erlauben und zahlreiche Fehler aufgedeckt haben.

1.5 Einführende Begriffe

Bevor auf die Details von formalen Sprachdefinitionen eingegangen werden kann, müssen einige häufig verwendete Begriffe erläutert werden.

1.5.1 Notationen, Konstrukte und Konzepte

Eine Computersprache wird in der Regel durch Symbole zu Papier oder auf den Computerbildschirm gebracht. Diese Symbole folgen einer Ordnung, die festlegt, welche Symbole in welcher Art und Weise gruppiert werden dürfen. Die Gesamtheit dieser Regeln heißt *Notation*. Eine Notation beschränkt sich stets auf die Regeln der Gruppierung von Symbolen und schließt also nicht eine bestimmte Bedeutung der Symbole ein.

Beispiel 1. Die Notation, mit der man Programme der Programmiersprache C aufschreibt, umfasst die Menge der verwendbaren Zeichen (im Wesentlichen die Groß- und Kleinbuchstaben, die Ziffern und einige Sonderzeichen des ASCII-Zeichensatzes), die Art und Weise, wie man aus diesen Zeichen Wörter bilden kann, sowie die Regeln, wie diese Wörter zu Übersetzungseinheiten zusammengefügt werden können.

Jedes Programm einer Computersprache besitzt eine Bedeutung. Der Leser eines Programms erkennt diese Bedeutung, indem er die Symbole, mit denen das Programm notiert ist, in Relation zu den *Konzepten* der Computersprache setzt. Ein Konzept ist eine abstrakte Idee; die unterstützten Konzepte einer Computersprache bestimmen ihre Ausdruckskraft.

Beispiel 2. Die Programmiersprache C unterstützt die Konzepte Funktion, Argument, Datentyp, Zuweisung, Variable, Ausdruck, und andere mehr.

Oft können innerhalb einer Notation separate Symbole oder Symbolgruppen einzelnen Konzepten zugeordnet werden. Solche Teile der Notation nennt man *Konstrukte*: Ein Konstrukt ist die Repräsentation eines Konzepts in einer bestimmten Notation. Viele Computersprachen verfügen über gleiche oder ähnliche Konzepte; sie unterscheiden sich aber in ihren Konstrukten.

Beispiel 3. Zuweisungen in C werden in etwa[2] durch die Symbolfolge

Variable = *Ausdruck*;

notiert, wobei *Variable* und *Ausdruck* Platzhalter für bestimmte andere Konstrukte sind, nämlich für Konstrukte, die die Notation der Konzepte Variable und Ausdruck bilden.

2. Tatsächlich weicht die Anweisungssyntax von der hier angegebenen ab: Zuweisungen sind zunächst Ausdrücke, keine Anweisungen, und auf der linken Seite dürfen beliebige L-Werte erscheinen, nicht nur Variablen.

1.5.2 Theorien und Kalküle

Grundlage der Formalisierung einer Computersprache ist eine mathematische *Theorie*, also eine Menge konsistenter Aussagen über einen mathematischen Objektbereich [Tar77]. Zu dieser Theorie gehört oft ein *Kalkül*, also ein Verfahren zur Manipulation mathematischer Objekte [Mat01]. Ein solcher Kalkül ermöglicht es, weitere Aussagen über die formale Sprachdefinition beispielsweise durch Anwendung von Schlussregeln zu gewinnen.

1.6 Typographische Konventionen

In dieser Arbeit wird eine Reihe von Konventionen benutzt, um auf besondere Bedeutungen mancher Wörter hinzuweisen. Kursivschrift wird verwendet, um bestimmte Begriffe zu *betonen* oder zu *definieren*, sowie um *englische* Begriffe hervorzuheben. Fettdruck zeigt **Schlüsselwörter** an. Durch serifenlose Schrift werden Bezeichner einer Computersprache ausgezeichnet. Absätze, die Programmtext enthalten, werden zudem eingerückt. Ebenfalls eingerückt werden Beispiele und *Zitate* (die zusätzlich geneigt geschrieben werden).

1.7 Abkürzungen

Eine Reihe häufig wiederkehrender Abkürzungen ist in Tabelle 1 erläutert.

Tabelle 1: Abkürzungen

Abkürzung	Bedeutung
ASM	Abstract State Machine
BNF	Backus-Naur-Form
ITU	International Telecommunication Union
ITU-T	ITU, Telecommunication Standardization
OMG	Object Management Group
SDL	Specification and Description Language
UML	Unified Modelling Language

2 Formale Definition von Sprachen und Notationen

Bei der Verarbeitung von Sprachen und Notationen durch eine Maschine ist es hilfreich, wenn diese Sprache formalen Regeln folgt. Wenn solche Regeln existieren, können Programme automatisch die Struktur und eventuell auch den Inhalt von Sätzen der Sprache „verstehen“, ohne über Hintergrundwissen (beispielsweise des Autors solcher Sätze) verfügen zu müssen.

In der Vergangenheit wurden verschiedene Formalismen verwendet, um solche Regeln zu erfassen, die auf unterschiedlichen Abstraktionen operieren. Um die formale Semantik von SDL in Relation zu der Formalisierung anderer Sprachen zu setzen, sollen hier einige gebräuchliche Techniken vorgestellt werden.

Die Sprachen und Notationen sollen im Folgenden zusammenfassend *Computersprachen* genannt werden; ein konkretes Dokument in einer Computersprache heißt dann *Computerprogramm*. Diese allgemeine Definition von Computersprachen, die über die gebräuchliche Definition auf Basis eines Alphabets hinausgeht, ist einer Charakteristik von SDL geschuldet: SDL kann nicht nur in der textuellen Form einer traditionellen Programmiersprache notiert werden, sondern auch mit grafischen Symbolen, die auf einer zweidimensionalen Fläche (einem Stapel Papier oder einem Computermonitor) gezeichnet werden.

Die in diesem Kapitel vorgestellten Techniken sind in der Vergangenheit für verschiedene Programmiersprachen verwendet worden. Der Leser dieser Arbeit wird einige der Techniken in der Definition von SDL wiederfinden und somit die Definition von SDL in das Gesamtbild formaler Definitionen von Sprachen und Notationen einordnen können. Die hier ausgewählten Beispiele haben unmittelbar nichts mit der Sprache SDL zu tun; sie illustrieren lediglich die verwendeten Techniken. Je höher der erreichte Formalisierungsgrad der Sprachdefinition ist, desto weniger vergleichbare Beispiele aus der Geschichte gibt es, da die Entwicklung einer formalen Sprachdefinition zeitaufwendig ist. Deshalb muss zur Illustration auf Beispiele verschiedener Sprachen zurückgegriffen werden: man findet keine einzelne Sprache, die alle Aspekte der Formalisierung auf die gleiche Weise vereint wie SDL.

2.1 Die Backus-Naur-Form

Zur Definition der Grammatik[3] wurde in Algol 58 [PS58] erstmalig die Backus-Normal-Form (BNF) verwendet [McG02], die für Algol 60 dann zur Backus-Naur-Form erweitert wurde. Diese Notation hat sich im Lauf der Zeit als geeignet für eine Vielzahl von Computersprachen herausgestellt, und wird auch für die Formalisierung verschiedener Aspekte dieser Sprachen eingesetzt.

Zunächst wird hier nur die Notation vorgestellt. Die Bedeutung dieser Notation hängt auch vom Kontext ihrer Anwendung ab. Tatsächlich bezeichnet man mit „Backus-Naur-Form“ heutzutage eine Vielzahl von Notationen, die sich geringfügig unterscheiden.

Ein BNF-Dokument besteht aus einer Folge von Regeln (oft *Produktionsregeln* genannt). Jede Regel hat eine linke Seite und eine rechte Seite, beide Seiten sind durch ein Trennzeichen (üblicherweise „::=“) von einander getrennt. Auf der linken Seite steht der Name eines sogenannten Nicht-Terminalsymbols, auf der rechten Seite stehen Namen von weiteren Symbolen (sowohl sogenannte Terminalsymbole als auch Nichtterminalsymbole) sowie Interpunktionszeichen. Übliche Interpunktionszeichen sind:

- der senkrechte Strich („|“),
- das Semikolon,
- eckige, runde und geschweifte Klammern sowie
- der Stern („*“).

3. siehe Abschnitt 2.2.3

Manche dieser Zeichen (insbesondere der Stern und eckige Klammern) gehören zu Konstrukten, die in der ursprünglichen Backus-Naur-Form nicht vorgesehen waren. Solche Notationen werden dann oft als erweiterte BNF (EBNF) bezeichnet.

Mit dieser Notation wird üblicherweise eine feste Bedeutung assoziiert. Der senkrechte Strich bezeichnet Alternativen, das Semikolon beendet eine Regel, runde oder geschweifte Klammern gruppieren Symbolfolgen, eckige Klammern drücken einen optionalen Teil aus und der Stern (Kleene-Stern) bezeichnet beliebige Wiederholung des vorangegangenen Elements.

Beispiel 4. In der Java-Sprachdefinition [GJSB00] wird eine Variante der BNF verwendet, bei der die Regel für Klassendeklarationen die folgende Form hat

ClassDeclaration:

ClassModifiers$_{opt}$ class *Identifier Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

In der Java-Sprachdefinition wird von der traditionellen Verwendung abgewichen. Optionale Teile werden hier durch ein tiefgestelltes „opt“ gekennzeichnet. In der Notation von [ML86] würde diese Regel

```
ClassDeclaration ::= [ClassModifiers] "class" Identifier [Super] [Interfaces] ClassBody
```

lauten.

2.2 Definition einer formalen Syntax

Ursprünglich wurden Computerprogramme vorwiegend als Texte festgehalten, zumindest, wenn ihre direkte Verarbeitung durch den Computer gewünscht war. Auch gegenwärtig sind Programme überwiegend textuell notiert. Grafische Notationen werden üblicherweise zur Kommunikation zwischen Menschen verwendet.

Im Fall eines Programmtexts kann man das Programm als eine lineare Folge von Zeichen betrachten, die nach bestimmten Regeln gruppiert werden. Will man diese Regeln formalisieren, muss man folgende Festlegungen treffen:

- Was ist die Menge der gültigen Zeichen?
- Wie werden die Zeichen zu Wörtern zusammengefasst?
- Welche Folgen von Wörtern sind zulässig?

Diese Regeln zusammen bilden die *Syntax* einer Sprache. Es ist nicht zwingend erforderlich, zwischen der Bildung von Wörtern (der sogenannten Lexik) und der Zusammenfassung von Wörtern zu Sätzen der Sprache (der Grammatik) zu trennen, diese Trennung hilft aber beim Verständnis der Syntax und ist weit verbreitet [ASU86].

2.2.1 Definition des Zeichensatzes

Um festzulegen, welche Zeichen in einem Computerprogramm verwendet werden dürfen, bedient man sich üblicherweise des Begriffs vom *Alphabet* [ASU86]. Dieser Begriff bezeichnet eine Menge von Zeichen, die, auf Papier gedruckt oder am Bildschirm dargestellt, unterschiedlich aussehen sollten. Tatsächlich wird der Computer bei der Verarbeitung des Programms sich jedoch nicht am grafischen Aussehen der Zeichen orientieren, sondern auf einer maschinennahen Repräsentation von Zeichen, beispielsweise als Bytes, operieren.

In der Vergangenheit ist bei der Definition von Sprachen der Aspekt der internen Repräsentation der Zeichen im Computer oft ignoriert worden, teilweise bewusst, oft aber unabsichtlich. Stattdessen wurde angenommen, dass die Zeichen in der auf dem konkreten Computersystem üblichen Art und Weise kodiert wurden. Da bei der Definition der Sprache keine Annahmen über das Computersystem gemacht werden können, sind zwei Strategien erkennbar:

- Zum einen wird von einem minimalen Satz von verfügbaren Zeichen ausgegangen, wie er

durch verbreitete Normen garantiert wird. So wird beispielsweise für C [ISO9899] und C++ [ISO14882] ein Basiszeichensatz formuliert, der sich an ASCII [ISO646] anlehnt.

- Zum anderen wird bei der Definition großzügig Gebrauch von üblichen mathematischen oder neu erfundenen Symbolen gemacht, und es dem Entwickler von Werkzeugen überlassen, diese Symbole intern zu repräsentieren und am Bildschirm oder auf Papier darzustellen. Ein Beispiel für eine solche Sprache ist APL [ISO8485], oder Java [GJSB00], welches eine große Zahl von Zeichen im Quelltext zulässt, da es auf dem Unicode-Standard basiert [ISO10646].

Diese Techniken unterliegen bei den verschiedenen Computersprachen Variationen, so ist beispielsweise bei der Formalisierung von C in Form eines internationalen Standards festgestellt worden, dass der zugrundeliegende Standard ISO 646 eine Reihe von Zeichen, die in C zuvor üblich waren, gar nicht unterstützt (dazu gehören etwa die eckigen Klammern). Daraufhin wurde die Sprache erweitert, indem gewisse Zeichenfolgen des Basisalphabets als Ersatzdarstellungen eventuell fehlender Zeichen verwendet werden können. So steht etwa in C die Zeichenfolge ??(für das Zeichen [. Die Sprachdefinition von C enthält also einen Transformationsschritt, indem diese Folgen von Zeichen intern durch ein einziges ersetzt werden [ISO9899, 5.1.1.2].

Bevor allerdings derartige Transformationen durchgeführt werden können, muss überhaupt erst einmal eine Zuordnung von Bytefolgen in einer Quelltextdatei zu Zeichenfolgen erfolgen. Diese Zuordnung erfolgt üblicherweise durch Festlegung eines kodierten Zeichensatzes (*coded character set*, [ISO2022]), Da auf verschiedenen Computersystemen verschiedene Zeichensätze in Verwendung sind, ist es üblich, in Sprachdefinitionen keinen bestimmten vorzuschreiben, sondern diese Festlegung der Implementierung der Sprache zu überlassen.

Die Festlegung des Alphabets wurde bei keiner der untersuchten Computersprachen formalisiert. Stattdessen ist es üblich, wie oben angedeutet, das Alphabet durch Verweis auf eine andere Norm festzulegen, in der die Zeichen und ihre Bedeutung oft in tabellarischer Form angegeben sind.

2.2.2 Definition der Lexik

Bei der Definition der Lexik einer Computersprachen unterscheidet man oft zwischen verschiedenen Klassen von Zeichenfolgen (sogenannte *Tokenklassen*). Die einzelnen Elemente einer solchen Klasse heißen dann *Wörter* (*Token*). Üblich ist die Unterscheidung in die Klassen

1. Bezeichner: Sie werden verwendet als Namen von Variablen, Funktionen, Typen, u.s.w.
2. Schlüsselwörter: obwohl sie oft die Form von Bezeichnern haben, nehmen sie in der Sprache eine Sonderstellung ein, und sind nicht als Bezeichner verwendbar (u.U. entscheidet allerdings die Stellung des Bezeichners im Programm, ob es sich um ein Schlüsselwort handelt).
3. Freiraum: Gewisse Zeichenfolgen werden zur optischen Strukturierung des Programms verwendet. Sie tragen (außer als Trennzeichen) letztlich nicht zur Semantik des Programms bei. Bei vielen Sprachen fallen auch Kommentare in diese Klasse.
4. Interpunktionszeichen: Viele Sprachen besitzen eine feste Menge von vorab festgelegten Zeichenfolgen, die ihrer Form nach nicht Bezeichner sind, und wie Schlüsselwörter eine spezielle Bedeutung haben. Es ist üblich, dass es sich um einzelne Zeichen handelt, allerdings findet man auch Kombinationen von Zeichen (wie beispielsweise die Folge :: in C).

Es ist üblich, weitere Klassen von Zeichenfolgen zu definieren (etwa Zahlen- und Zeichenketten-Literale); für die Diskussion hier reichen aber diese Beispiele aus.

Es ist oft einfach, die Regeln der einzelnen Klassen in Worte zu fassen: Die Klassen 2 und 4 enthalten in der Regel nur endlich viele Elemente, die man durch Aufzählung alle formulieren

kann. Die Zeichenfolgen der Klassen 1 und 3 genügen einfachen Bildungsvorschriften. Üblich ist es etwa, für Bezeichner die Regel

Ein Bezeichner ist ein Buchstabe gefolgt von Buchstaben oder Ziffern.

anzugeben. Diese Definition ist natürlich unvollständig, da nun festgelegt werden muss, welche Zeichen des Alphabets denn als Ziffern und welche als Buchstaben gelten sollen. Auch ist nicht klar, ob die Zeichenfolge „foobar“ ein Bezeichner ist (foobar) oder vielleicht zwei („foo“ gefolgt von „bar“).

Um diese Schwierigkeiten zu lösen, verwendet man eine formale Grammatik, die eine Folge von Zeichen des Quelltexts einer Klasse zuordnet, wenn die Produktion dieser Zeichenfolge aus einem Hilfssymbol der Grammatik [ASU86] möglich ist. Beispielsweise könnte die Definition der angegebenen Bezeichnerregel lauten

```
bezeichner ::= buchstabe { buchstabe | ziffer }*
buchstabe ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Zwar löst diese Notation die Frage der exakten Definition jeder Klasse, gleichzeitig verschärft sich aber das Problem der Mehrdeutigkeit: Gewisse Zeichenfolgen fallen in mehrere Klassen. Wenn beispielsweise die Klasse der Schlüsselwörter definiert ist durch

```
schlüsselwort ::= begin | end | program | while
```

dann fällt die Zeichenfolge „while“ sowohl in die Klasse schlüsselwort als auch in die Klasse bezeichner. Dieses Problem ist mit Hilfe der BNF nicht einfach zu lösen, deshalb greift man oft zusätzlich auf Erklärungen in einer natürlichen Sprache zurück. Beispielsweise heißt es in der C++-Norm [ISO14882, 2.11]

The identifiers shown [...] are reserved for use as keywords [...]

Mit dieser Formulierung wird einer bestimmten Token-Klasse Vorrang vor einer anderen gewährt. Selbst mit dieser Formulierung sind die Mehrdeutigkeiten nicht beseitigt, da immer noch eine Folge von Buchstaben als einzelnes oder als mehrere bezeichner verstanden werden könnte. Dies wird in C++ durch die Formulierung

> *If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.*

gelöst.

Hier zeigen sich die Grenzen der Formalisierbarkeit mittels EBNF: Während die erste dieser Regeln (Schlüsselwörter haben Vorrang) noch in der EBNF erfasst werden könnte, lässt sich die zweite Regel nicht mehr durch Umformung der Grammatik fassen.

2.2.3 Definition der Grammatik

Mit Hilfe von BNF-Beschreibungen kann man bekanntlich kontextfreie Sprachen definieren [ASU86]. Seit der Verwendung von BNF zur Definition von Algol ist es üblich, die Grammatik von Sprachen mit Hilfe von BNF zu beschreiben, so verwenden die Sprachdefinitionen von C, C++ und Java spezifische Varianten der BNF.

In der Definition der Grammatik unterscheidet man dann zwischen Hilfssymbolen und Terminalsymbolen. Terminalsymbole entsprechen Tokenklassen oder einzelnen Wörtern; Hilfssymbole sind alle Symbole, die durch eine Grammatikregel definiert werden. Ein Hilfssymbol wird als *Startsymbol* der Grammatik ausgezeichnet.

Die Token der Lexik werden allerdings oft nicht vollständig Terminalsymbolen zugeordnet. In vielen Sprachen werden in der Grammatik manche Tokenklassen ignoriert, so zum Beispiel

Freiraum. So ist es beispielsweise in C erlaubt, zwischen beliebigen Wörtern Freiraum einzufügen. Anstatt dies in der Grammatik festzuhalten, wird in natürlicher Sprache formuliert, dass nach Zerlegung der Eingabe in Token-Folgen gewisse Wörter entfernt werden.

Die mit Hilfe der BNF-Beschreibung definierte Sprache soll im folgenden die *akzeptierte Sprache* heißen. Sie ist definiert durch die Menge aller Sätze, die sich durch wiederholte Anwendung von Produktionsregeln aus dem Startsymbol generieren lassen. Ein Satz der Sprache entsteht dann durch folgenden Produktionsprozess in Umkehrung des Analyseprozesses:

1. Aus dem Startsymbol wird durch wiederholte Anwendung von Produktionsregeln eine Folge von Terminalsymbolen gewonnen.
2. Für jedes Terminalsymbol, das zu einer Tokenklasse gehört, wird ein einzelnes Token als Repräsentant gewählt.
3. Entsprechend der Regeln zur Entfernung von Wörtern können löschbare Wörter (beispielsweise Freiraum) eingefügt werden. Dabei ist zu beachten, dass die Regeln der Lexik eventuell zwingend verlangen, an bestimmten Stellen solche Wörter einzufügen: Der Produktionsprozess ist nur dann richtig, wenn bei Zerlegung der Buchstabenfolge zurück in Wörter sich wieder die gleiche Wörterfolge ergibt.

2.2.4 Definition einer grafischen Syntax

Für Sprachen, die eine grafische Syntax besitzen, gibt es keine etablierten Kalküle zur formalen Definition ihrer Syntax. Die Schwierigkeit, ein solches Kalkül zu schaffen, liegt darin, dass ein Programm der Sprache nicht aus einer linearen Folge von Zeichen besteht, und deshalb der Kalkül kontextfreier Grammatiken nur begrenzt einsetzbar ist.

Beispiele für Sprachen, deren grafische Syntax trotzdem durch eine Grammatik beschrieben wurde, sind SDL und GLOTOS [ISO8807]. Da das Aneinanderfügen von Symbolen und Hilfssymbolen, das in BNF implizit in den Regeln enthalten ist, auf grafische Notationen nicht direkt anwendbar ist, wurden in der Grammatikformulierung von SDL einige zusätzliche Operatoren verwendet:

- **set** (Postfix-Operator): Das vorangehende Symbol kann mehrfach wiederholt werden, in beliebiger grafischer Anordnung.
- **contains** (Infix-Operator): Das Symbol auf der linken Seite des Operators umschließt das Symbol auf der rechten Seite.
- **is associated with** (Infix-Operator): Der rechte Operand wird in der Nähe des linken Operanden dargestellt, so dass die Symbole als zusammengehörig erkannt werden.
- **is followed by** (Infix-Operator): Zwischen dem Symbol auf der linken und der rechten Seite wird eine Kontrollflusslinie gezeichnet, die am unteren Ende des ersten Symbols beginnt und am oberen Ende des zweiten Symbols endet.
- **is connected to** (Infix-Operator): Die Symbole berühren sich.

In anderen Sprachen begnügt man sich damit, die grafische Syntax mit natürlicher Sprache und durch Angabe der Symbole zu beschreiben. So wird für UML [OMG01] ein *Notation Guide* definiert, wo für jedes Konzept von UML eine entsprechende Symbolik abgebildet ist und mit englischen Worten erklärt wird, wie diese Symbolik verwendet werden soll.

2.3 Statische Semantik und Wohlgeformtheit

In der Regel wird eine kontextfreie Grammatik die Formulierung von Sätzen in der Sprache erlauben, die eigentlich als ungültig gelten sollen. So entspricht beispielsweise in C++ das Programm

```
int main()
{
    return Ergebnis;
}
```

durchaus der Grammatik von C++, ist jedoch trotzdem ungültig, da die Variable Ergebnis nicht definiert ist.

In manchen Fällen kann man versuchen, die Regeln der Grammatik so zu verändern, dass weniger als ungültig zu betrachtende Programme der Grammatik entsprechen. Für die meisten Sprachen kann das aber nicht vollständig gelingen, da, wie im Beispiel gezeigt, Aussagen über die Gleichheit von Token zu treffen sind, obwohl in der Grammatik als Terminalsymbol nur eine Tokenklasse auftritt.

Die Einschränkungen der akzeptierten Sprache auf gültige Sätze erfolgt oft mit Hilfe von Regeln, die Bezug auf die Bedeutung des Satzes nehmen. So ist das C++-Beispiel falsch, weil es die Regel

> *Name lookup associates the use of a name with a declaration (3.1) of that name. Name lookup shall find an unambiguous declaration for the name (see 10.2).*

verletzt: Für den Namen Ergebnis findet der Prozess des *name lookup* keine Deklaration des Namens. Mit anderen Worten: Ergebnis ist ein undefinierter Bezeichner.

Tatsächlich definieren solche Regeln allerdings nicht die Bedeutung des Programms, zumindest nicht auf eine formal erfassbare Weise: sie schränken lediglich die akzeptierte Sprache weiter ein. Implizit enthalten die Regeln sicherlich auch die Bedeutung des Programms. So ist dem Leser sicherlich klar, dass die Regel auch beschreibt, dass der Name, wenn er im „name lookup“ eine eindeutige Deklaration gefunden hat, diese Verwendung des Namens sich auf die Deklaration bezieht. Dieser Zusammenhang trägt allerdings erst dann zur Bedeutung des Programms bei, wenn man die Abarbeitung der Funktion „main“ betrachtet: Dann nämlich muss die Auswertung des Ausdrucks „Ergebnis“ Bezug nehmen auf den Wert, den die Variable Ergebnis in diesem Moment hat.

So ist es also wichtig zu unterscheiden, ob eine bestimmte Regel lediglich die Menge gültiger Sätze der Sprache einschränkt, oder dem Programm tatsächlich auch eine Bedeutung gibt, die bei der Abarbeitung des Programms sichtbar wird.

Die Regeln der ersten Art sollen im Folgenden *statische Semantik* heißen, es ist auch üblich, sie Wohlgeformtheitsregeln (*well-formedness rules*) zu nennen, da sie aus der akzeptierten Sprache diejenigen Sätze auswählen, die tatsächlich als „richtig“ oder „wohlgeformt“ gelten sollen. Regeln, die die Abarbeitung des Programms betreffen, bilden zusammen die *dynamische Semantik*.

Zur Formalisierung sowohl von statischer als auch dynamischer Semantik wurden in der Vergangenheit zahlreiche Kalküle eingesetzt. Im folgenden sollen einige derjenigen Kalküle vorgestellt werden, die auch tatsächlich in der Normierung von Sprachen Verwendung fanden. Tatsächlich lassen sich nicht sehr viele solcher Beispiele finden – insbesondere die Formulierung der statischen Semantik erfolgt meist nur informal.

2.3.1 Algol-68 und zweistufige Grammatiken

Zur Definition von Algol-68 [FKSH72] wurden sogenannte van-Wijngaarden-Grammatiken verwendet (auch zweistufige Grammatiken genannt). Mit diesem Kalkül können nicht nur syntaktische Eigenschaften eines Programms definiert, sondern auch semantische Zusammenhänge festgehalten werden. Dabei gibt es zwei Arten von Grammatikregeln: Grammatikregeln für eine Grundsprache, und Grammatikregeln für eine Metasprache. Grammatikregeln für die Metasprache sind Regeln, in denen Platzhalter die Generierung von Regeln der Grundsprache erlauben.

Beispiel 5. Die Metasprache von Algol-68 enthält die Regel

```
chain of NOTIONs separated by SEPARATORs:
        NOTION;
        NOTION, SEPARATOR, chain of NOTIONs separated by SEPARATORs.
```

Damit wird ein Metabegriff definiert, der mit zwei Argumenten (NOTION und SEPARATOR) parametrisiert ist. In dieser Form der BNF trennt das Semikolon Alternativen; das Komma trennt Symbole einer Alternative. Der Doppelpunkt trennt den Namen des definierten Hilfssymbols von seiner Definition, und der Punkt beendet die Regel. Inhaltlich bezeichnet dieser Metabegriff eine Folge von Dingen, die durch einen Trenner getrennt werden. Auf der Basis dieses Metabegriffs werden andere Begriffe definiert, etwa

```
statement prelude:
    chain of strong void units separated by go on symbols, go on symbol.
```

Dabei sind strong void unit und go on symbol wiederum Namen von Regeln. In statement prelude wechseln also strong void unit und go on symbol einander ab. Die Dinge sind strong void units, der Trenner ist das go on symbol. Durch die Einführung eines Metabegriffs wurde hier also ein häufig wiederkehrendes Muster formalisiert, von welchem dann in verschiedenen Regeln (wie etwa statement prelude) Gebrauch gemacht wird. Es handelt sich bei diesem Metasymbol also um eine Erweiterung der Ausdrucksmöglichkeiten der BNF, die ähnlich ist zum Kleene-Stern. Ohne diese Metaregel hätte statement prelude wie folgt definiert werden müssen:

```
statement prelude:
    strong void unit, go on symbol;
    strong void unit, go on symbol, statement prelude.
```

Durch Metaregeln können nicht nur Grammatikregeln erweitert werden, sondern auch syntaktische Einschränkungen vorgenommen werden. Die Metaregeln erlauben die Generierung einer beliebig großen Menge von Grammatikregeln der Grundsprache. Damit können durch die Grammatik Einschränkungen formuliert werden, die mit endlich vielen Produktionsregeln nicht ausgedrückt werden können.

Folgendes Beispiel demonstriert, wie mit Hilfe von Metaregeln semantische Einschränkungen festgelegt werden können.

Beispiel 6. Der logische Vergleich auf Identität wird durch die Grammatikregel

```
boolean identity relation:
    soft reference to MODE tertiary, identity relator, strong reference to MODE tertiary;
    strong reference to MODE tertiary, identity relator, soft reference to MODE tertiary.

identity relator: is symbol; is not symbol.
```

definiert. Dabei ist MODE ein Platzhalter für einen Modus (also einen Datentyp); die Metaregel soft reference to MODE tertiary lässt sich nur zu Ausdrücken expandieren, deren Typ MODE ist.

Diese eine Regel boolean identity relation definiert eine beliebig große Zahl von Regeln in „traditioneller“ BNF, etwa

boolean identity relation:
soft reference to **int** tertiary, identity relator, strong reference to **int** tertiary;
strong reference to **int** tertiary, identity relator, soft reference to **int** tertiary:

soft reference to **bool** tertiary, identity relator, strong reference to **bool** tertiary;
strong reference to **bool** tertiary, identity relator, soft reference to **bool** tertiary:

...

soft reference to **real** tertiary, identity relator, strong reference to **real** tertiary;
strong reference to **real** tertiary, identity relator, soft reference to **real** tertiary.

Wenngleich die Meta-Regel die Definition von beliebig vielen Regeln erlaubt, so haben diese Regeln doch alle eines gemein: Der Datentyp links und rechts von identity relator ist immer der gleiche.
Da verlangt wird, dass auf beiden Seiten von boolean identity relation der gleiche Modus auftreten muss, legt diese Regel nicht nur fest, dass ein Identitätsvergleich zwei Operanden hat, sondern auch, dass diese Operanden den gleichen Modus haben müssen und einer eine starke und der andere eine schwache Referenz sein muss. Die Eigenschaft, starke oder schwache Referenz zu sein, gibt dabei die Möglichkeiten von Typumwandlungen an. Auch diese Typumwandlungen sind in Form von Metaregeln definiert. Insgesamt definiert die Regel also nicht nur, dass

x :=: y

den Identitätsvergleich der Variablen x und y durchführt, sondern auch, dass diese Variablen den gleichen Typ haben müssen.

2.3.2 UML und die Object Constraint Language

Mit der Sprache UML (*Unified Modeling Language*) können Teile[4] eines Softwaresystems auf grafische Weise visualisiert, spezifiziert, konstruiert und dokumentiert werden [OMG01]. Programme der Sprache UML werden *Diagramme* genannt. Teil dieser Sprache ist die Sprache OCL (*Object Constraint Language*), mit der seiteneffektfreie Bedingungen für UML-Diagramme definiert werden können. Durch OCL-Ausdrücke können Prädikate für gewisse Symbole definiert werden, die für diese Symbole eines Diagramms erfüllt sein müssen.

Die Sprache UML selbst ist durch Angabe ihrer Konzepte und Notationen definiert. Die Konzepte bilden zusammen ein *Metamodell*.

Dieses Metamodell ist wiederum durch UML-Diagramme definiert, in denen den UML-Konzepten Namen gegeben ("Namespace", "Method" u.s.w.) und Relationen zwischen diesen Konzepten definiert werden. Diese Diagramme bilden zusammen die *abstrakte Syntax* von UML. Die Syntax heißt abstrakt, weil sie strukturelle Zusammenhänge zwischen den Konzepten repräsentiert, ohne auf die tatsächliche Gestaltung der Symbole einzugehen. Abbildung 1 stellt einen Ausschnitt des UML-Metamodells dar. Die konkreten Notationen für UML werden in englischer Sprache und durch Abbildungen erläutert.

Da für die Elemente der abstrakten Syntax Einschränkungen gelten sollen, die sich nicht durch die abstrakte Syntax erfassen lassen, sind neben der Definition der abstrakten Syntax eine Reihe von OCL-Prädikaten definiert worden. Ein korrektes UML-Programm muss nicht nur der abstrakten Syntax folgen, sondern auch sämtliche Prädikate erfüllen.

4. [OMG01] spricht von *artefacts*.

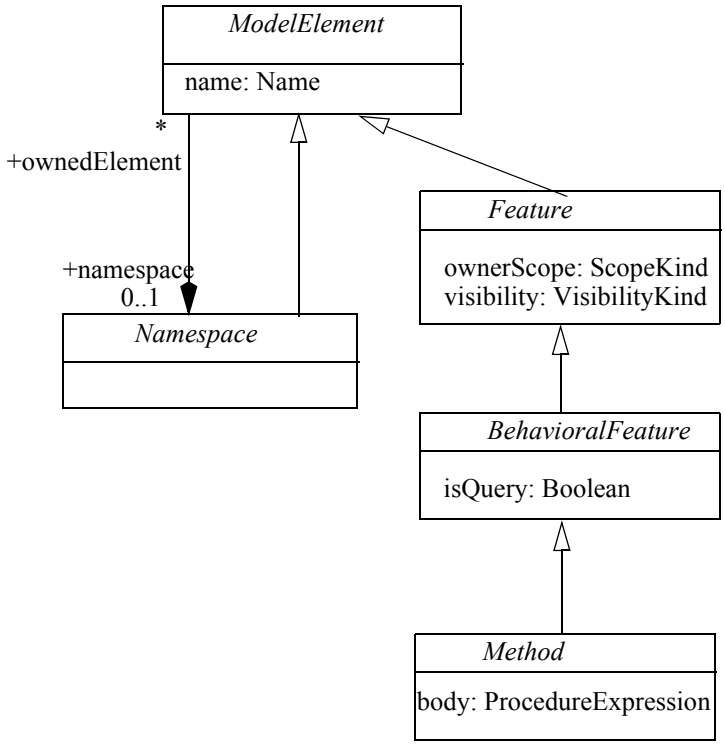

Abbildung 1: Ausschnitt des UML-Metamodells

Beispiel 7. Für das Konzept Component wurde in UML folgende zusätzliche Forderung aufgestellt:

A Component may only have as residents DataTypes, Interfaces, Classes, Associations, Dependencies, Constraints, Signals, DataValues, and Objects.

Folgender OCL-Ausdruck formalisiert diese Forderung:

```
self.allResidentElements->forAll(
    re |
    re.oclIsKindOf(DataType) or
    re.oclIsKindOf(Interface) or
    re.oclIsKindOf(Class) or
    re.oclIsKindOf(Association) or
    re.oclIsKindOf(Dependency) or
    re.oclIsKindOf(Constraint) or
    re.oclIsKindOf(Signal) or
    re.oclIsKindOf(DataValue) or
    re.oclIsKindOf(Object) )
```

Dabei ist self ein Ausdruck, der die zu überprüfende Komponente bezeichnet. Diese besitzt ein Attribut allResidentElements, das die Menge aller enthaltenenen Modellelemente enthält. Diese Menge (wie jede andere OCL-Menge auch) unterstützt eine Methode forAll, die ein all-quantifiziertes Prädikat berechnet, welches in Klammern angegeben ist.

2.4 Dynamische Semantik

Ziel der Entwicklung eines Computerprogramms ist es typischerweise, dieses von einem Computer abarbeiten zu lassen. Die Bedeutung des Programms besteht dann in der Folge der Aktionen, die der Computer ausführt. Diese ist auch von der Umgebung (etwa der Folge von Eingabeaktionen) abhängig. Die Regeln, die für eine Computersprache beschreiben, auf welche Weise die Abarbeitung erfolgt, bilden zusammen die *dynamische Semantik* der Sprache.

Hierbei kann man zwei Arten von Festlegungen beobachten:

- Festlegung von funktionalen Aspekten, etwa in Form von beobachtbaren Interaktionen des Computers mit seiner Umgebung und
- Festlegung von nicht-funktionalen Aspekten, etwa durch Angabe von maximalen Abarbeitungszeiten oder maximalem Hauptspeicherverbrauch.

Im Rahmen dieser Arbeit liegt der Schwerpunkt auf funktionalen Aspekten, deshalb sollen im Folgenden ein Kalkül vorgestellt werden, das bei der Normierung von Computersprachen Verwendung gefunden hat, nämlich bei der Definition von Scheme. Tatsächlich gibt es nur wenige Beispiele von Sprachen, für die eine formale dynamische Semantik Teil des Sprachstandards ist. Die Kalküle, die zur Definition von SDL eingesetzt wurden, werden in Kapitel 5 dargestellt.

Für die funktionale Programmiersprache Scheme wurde in [KCR98] eine denotationale Semantik [Sto77] definiert. Diese formale Definition ist relativ klein (zwei Seiten Text). Dies liegt vor allem daran, dass es in Scheme nur wenig fundamentale Konstrukte gibt, deren Semantik definiert werden muss (nämlich Bezeichner, Konstanten und Ausdrücke).

In einer denotationalen Semantik wird einem Programm eine Bedeutung in Form mathematischer Objekte zugeordnet. Für Scheme werden mathematische Objekte für die Repräsentation des Scheme-Programms, für natürliche Zahlen, Lokationen, Speicher, Fortsetzungen und andere Domänen definiert. Dabei sind Fortsetzungen Objekte, die einen Berechnungsschritt repräsentieren.

Eine Reihe von Funktionen definiert dann die Bedeutung eines Scheme-Programms, indem für jedes Scheme-Konstrukt eine Formel angegeben wird, die eine Fortsetzung für dieses Konstrukt ermittelt. Die Bedeutung des Gesamtprogramms ergibt sich dann, indem die Fortsetzungen aller Konstrukte aneinandergereiht werden. Wendet man diese Fortsetzung auf den Initialzustand der Interpretation an, so erhält man das Ergebnis der Berechnung.

2.5 Über die Rolle von Werkzeugen

Durch eine formale Sprachdefinition kann die Exaktheit und Klarheit der Definition erhöht werden: die Formeln definieren unzweideutig, was im englischen Text vielleicht mehrdeutig ist.

Allerdings führt eine formale Definition auch zu neuen Problemen: Die formale Definition kann unvollständig, widersprüchlich oder falsch sein. Beispielsweise könnten essentielle Teile der Definition fehlen, wie Regeln der Grammatik oder Funktionen einer denotationalen Semantik fehlen, oder es können Bedingungen für die statische Semantik definiert sein, die sich gegenseitig ausschließen. In diesen Fällen verliert die formale Definition ihre Vorteile. Der Leser ist genauso auf seine eigene Intuition angewiesen wie beim Studium der informalen Sprachdefinition.

Im Prinzip kann man die Abwesenheit solcher Fehler beweisen. Ein Beweis wäre aber meist länger als die eigentliche formale Semantikdefinition und könnte selbst wieder fehlerhaft sein.

Viele Probleme der formalen Sprachdefinition lassen sich mit Hilfe von Werkzeugen vermeiden. Werkzeuge können die Vollständigkeit der Definition überprüfen, indem sie feststellen, ob alle verwendeten mathematischen Objekte auch definiert wurden. Für Probleme, die sich nicht durch komplette automatische Analyse entdecken lassen, können sie das Testen unterstützen: Durch Ausführung der formalen Definition unter Eingabe von Beispielprogrammen kann ermit-

telt werden, ob die formale Definition für die eingegebenen Programme die erwarteten Ergebnisse liefert.

Neben den Problemen, die der Autor einer formalen Definition zu lösen hat, stellen sich auch für den Leser der Sprachdefinition neue Herausforderungen: Er muss sich zum Verständnis der Sprachdefinition mit den verwendeten Formalismen auseinandersetzen. Auch hierbei können Werkzeuge helfen: Das Werkzeug kann in einem Blackbox-Modus eine Referenzimplementierung der Sprache bereitstellen, bei der der Leser eigene Beispiele an das Werkzeug übergibt und die Abarbeitungsergebnisse studiert, ohne sich mit den Verfahren der Abarbeitung zu beschäftigen. Alternativ kann das Werkzeug einen Whitebox-Modus bereitstellen, in dem der Anwender die einzelnen Verarbeitungsschritte inspizieren und mit seinem Verständnis der Computersprache vergleichen kann.

2.6 Fazit

In diesem Kapitel wurden verschiedene Techniken vorgestellt, die in der Vergangenheit zur formalen Definition von Computersprachen eingesetzt wurden. Diese Techniken illustrieren die Aspekte, die in einer formalen Sprachdefinition beachtet werden müssen. Einige dieser Techniken werden auch zur Definition der SDL-Semantik verwendet (insbesondere zur Definition von Grammatiken). Für andere Aspekte von SDL wurden spezielle Techniken geschaffen, die in Kapitel 5 vorgestellt werden.

3 SDL-2000

SDL (*Specification and Description Language*) ist eine Sprache zum Entwurf und zur Beschreibung von reaktiven Systemen [Z.100-00]. Obwohl sie für die Telekommunikationsindustrie entwickelt wurde, wird SDL auch in anderen Zweigen der Industrie verwendet. Auf dem Gebiet der Telekommunikation kann SDL unter anderem für die Entwicklung folgender Anwendungen eingesetzt werden:

- Verarbeitung von Anrufen und Verbindungen in Vermittlungssystemen,
- Betrieb und Fehlerbehandlung in allgemeinen Telekommunikationssystemen,
- Systemkontrolle,
- Datenkommunikationsprotokolle und
- Telekommunikationsdienste.

Allgemein kann SDL für die funktionale Beschreibung sämtlicher Systeme eingesetzt werden, die durch ein diskretes Modell beschrieben werden können, wo also Objekte über diskrete Nachrichten kommunizieren. Die Nachrichten heißen *diskret*, weil sie zu bestimmten Zeitpunkten erzeugt werden und also keine kontinuierlichen Informationen darstellen.

SDL wird in verschiedenen Phasen der Entwicklung solcher Systeme eingesetzt, unter anderem für die Entwicklung von

- Anforderungsbeschreibungen,
- Systemspezifikationen,
- ITU-T-Empfehlungen,
- Systementwurfsbeschreibungen und
- Systemtestbeschreibungen.

Die Entwicklung von SDL begann 1972; 1976 wurde die erste Version der Sprache veröffentlicht [Z.100-00]. Die wesentlichen Meilensteine der Entwicklung sind in Tabelle 2 vorgestellt.

Tabelle 2: Entwicklung von SDL

Jahr	Wesentliche Neuerungen
1976	Erste Sprachversion
1980	Einführung hierarchischer Strukturierungsmöglichkeiten
1984	Einführung abstrakter Datentypen
1988	Definition einer formalen Semantik
1992	Definition objekt-orientierter Konzepte, Nichtdeterminismus, *Remote Procedure Calls* (RPC)
1995	SDL in Kombination mit ASN.1 (Z.105)
1996	SDL-Austauschformat (Z.106)
1996	Korrekturen und Vereinfachung (Harmonisierung von Konzepten) von SDL-92
1999	SDL-2000: Syntaktische Anpassung an andere Sprachen; Harmonisierung der Konzepte System, Block, Prozess; Schnittstellen; Ausnahmebehandlung; textuelle Algorithmen; verschachtelte Zustände; objektorientierte Daten; neue Integration mit ASN.1

Tabelle 2: Entwicklung von SDL

Jahr	Wesentliche Neuerungen
2002	Korrekturen von SDL-2000; Definition der grafischen Syntax als „primäre“ konkrete Syntax

Im Folgenden werden einige Konzepte von SDL vorgestellt. Weiteres einführendes Material findet der Leser in [OFM+94] und [Dol01]. Die exakte Definition der Sprache ist in [Z.100-00] festgehalten.

3.1 Systeme

Ein SDL-Programm (auch *Spezifikation* genannt) definiert in der Regel ein *System*. Ausnahme sind Spezifikationen, die lediglich Pakete (Bibliotheken) enthalten. Für Systeme ist der Begriff der *Interpretation* (auch: *Abarbeitung*) definiert. Während der Interpretation nimmt das System verschiedene Zustände an. Die dynamische Semantik von SDL beschreibt, welche Folgen von Zuständen für ein gegebenes System während einer Interpretation potentiell auftreten können.

Da mit SDL auch verteilte Systeme beschrieben werden, muss berücksichtigt werden, dass bei Kommunikation zwischen den Teilen des Systems Verzögerungen auftreten können und die Teile mit verschiedenen Geschwindigkeiten arbeiten. Diese relativen Geschwindigkeiten sind nicht Teil der Spezifikation, statt dessen sind für ein System alle Interpretationen erlaubt, die sich durch unterschiedliche Abarbeitungsgeschwindigkeiten ergeben können.

3.2 Agenten

Agenten bilden die aktiven Komponenten eines Systems. Sie sind seit der Version SDL-2000 Teil der Sprache. In früheren Versionen der Sprache besaß lediglich eine Art der Agenten, die *Prozesse*, ein aktives Verhalten. In SDL-2000 besitzen auch die anderen Arten von Agenten (*Blöcke* und *Systeme*) Verhalten.

Jeder Agent wird durch einen erweiterten endlichen Zustandsautomaten beschrieben, wie er in Abbildung 2 dargestellt ist. Durch den äußeren Rahmen und das Schlüsselwort Process wird ein Prozess definiert. Die Zahlen 1(1) in der rechten oberen Ecke zeigen an, dass diese Seite die Seite 1 von insgesamt einer Seite (der Prozessdefinition) ist.

In dieser Abbildung wird ein Prozess-Agent Initiator definiert, der zwei Zustände (WAIT und Connected) besitzt. Durch die Definition einer weiteren Zustandsvariablen count kann der Prozess potentiell unendlich viele Zustände annehmen. Deshalb werden die Agenten als *erweitert endlich* bezeichnet.

Der Zustandsautomat des Agenten kann, wie im Beispiel, direkt im Rahmen des Prozesses eingezeichnet werden, oder aber separat durch Verwendung eines Zustandstypen. Falls für einen Agenten gar kein Zustandsautomat angegeben ist, wird implizit ein Automat angenommen, der in einem anonymen Zustand verharrt.

Agenten werden entweder bei Systemstart erzeugt oder aber dynamisch während der Abarbeitung des Systems. Nachdem ein Agent erzeugt wurde, beginnt er mit der Abarbeitung der Start-Transition, die durch das Start-Symbol (⬭) eingeleitet wird. Der Agent führt so lange Aktionen durch, bis er einen nächsten Zustand erreicht. Im Beispiel wird ein Signal ICONreq versendet, der Zeitgeber T gestartet und in den Zustand WAIT übergegangen.

Der Agent verharrt in einem Zustand so lange, bis er ein Signal oder einen anderen Stimulus erhält (beispielsweise bis ein Zeitgeber abläuft). Empfangene Signale werden in einem Puffer in der Reihenfolge ihres Eintreffens gespeichert, und der Agent konsumiert aus diesem Puffer

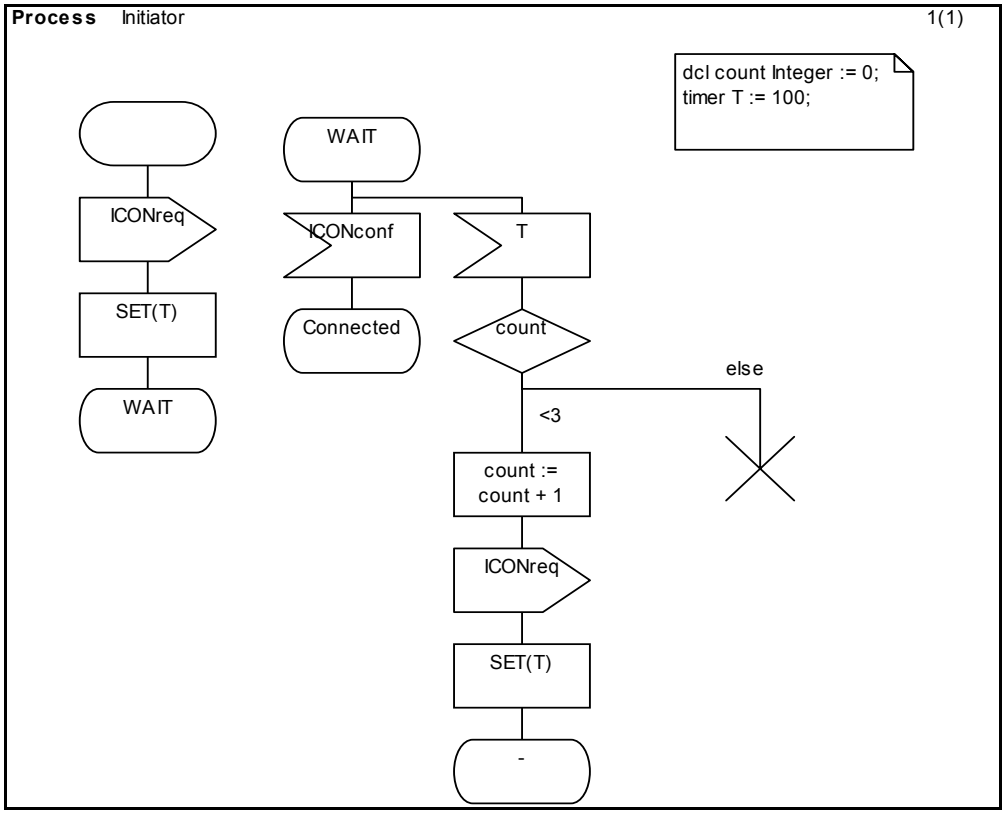

Abbildung 2: Zustandsgraph eines Prozessagenten

stets das erste Signal[5]. Ist für einen Zustand keine Transition für ein Signal festgelegt, wird das Signal verworfen.

Im Beispiel geht der Agent bei Empfang des Signals ICONconf in den Zustand Connected über. Bei Ablauf des Zeitgebers T überprüft er die Zahl der Verbindungsversuche count. Ist diese kleiner als 3, sendet er ein weiteres Signal, startet den Zeitgeber erneut, und erhöht die Variable count. Ansonsten (also beim vierten Versuch) terminiert der Agent, angezeigt durch das Stop-Symbol ($\times$).

Neben der Definition von Zuständen und Transitionen gibt es eine Reihe weiterer Strukturierungskonzepte für das Verhalten eines Agenten:

- Durch Definition von verschachtelten Zuständen (*state aggregation, state composition*) können zusammengehörende Transitionen gruppiert werden.
- Durch Definition von Prozeduren können Algorithmen vom eigentlichen Zustandsgraphen separiert und gegebenenfalls mehrfach verwendet werden.
- Durch Definition von Ausnahmebehandlungstransitionen kann das Verhalten im Fehlerfall vom normalen Verhalten getrennt werden.

3.3 Systemstrukur

Agenten können hierarchisch strukturiert sein, indem Agenten andere Agenten enthalten. Dabei gibt es drei Arten von Agenten: Systeme, Blöcke und Prozesse. In einer Spezifikation kann höchstens ein System definiert werden. Dieses System wird in einer Umgebung interpretiert, mit der es kommunizieren kann.

5. Von dieser Reihenfolge kann durch besondere Sprachmittel abgewichen werden.

Das System kann weitere Agenten enthalten, also Blöcke und Prozesse. Blöcke und Prozesse sind semantisch nahezu gleichbedeutend, mit folgenden Unterschieden:

1. Die in Blöcken (und im System) enthaltenen Agenten können gleichzeitig arbeiten. Von Agenten in Prozessen kann immer nur einer zu jedem Zeitpunkt aktiv sein (also eine Transition abarbeiten).
2. Blockagenten können sowohl Blöcke also auch Prozesse enthalten, Prozesse dürfen nur weitere Prozesse enthalten.

Ein Beispiel für einen strukturierten Block ist in Abbildung 3 angegeben.

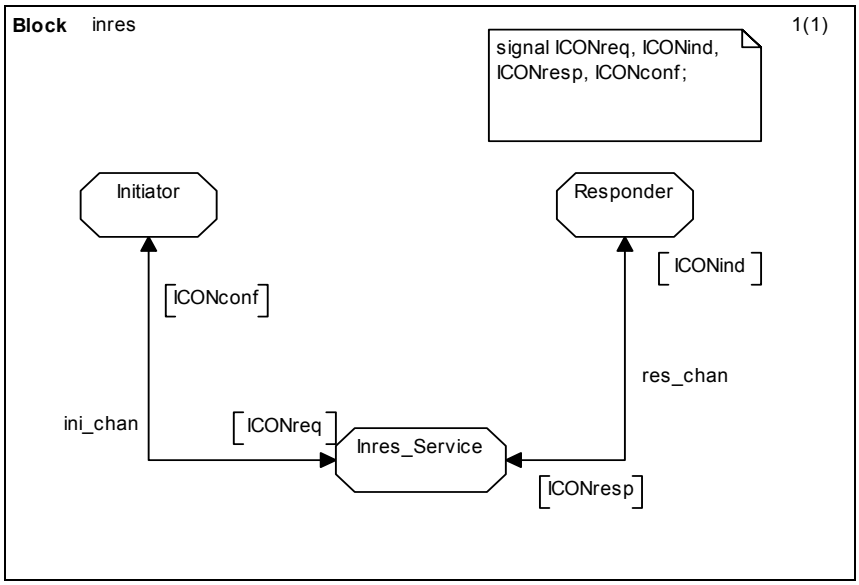

Abbildung 3: Struktur eines Blocks

In diesem Beispiel besteht der Block inres aus drei Prozessen: Initiator, Inres_Service und Responder. Diese Agenten kommunizieren über eine Reihe von Signalen, die innerhalb des Textsymbols definiert werden.

Zum Austausch der Signale werden zwischen den Prozessen zwei Kanäle ini_chan und res_chan vereinbart. Für jeden dieser Kanäle ist festgelegt, welche Signale er in welche Richtung transportieren kann. Aus den Kanaldefinitionen ergibt sich, welche Signale jeder Prozess senden und empfangen kann. In Abbildung 2 hat der Prozess Initiator das Signal ohne Angabe eines Empfängers versandt. Aus der Definition des Blocks inres ergibt sich nun, dass der Empfänger der Prozess Inres_Service ist. Falls der Empfänger eines Signals mehrdeutig ist, wird aus den möglichen Empfängern nicht-deterministisch einer ausgewählt. Ist diese Auswahl nicht erwünscht, so kann durch explizite Angabe des Empfängers in der Output-Aktion (⬠) ein Empfänger ausgewählt werden. Zu diesem Zweck verfügt jeder Agent über eine eindeutige Identifikation, die *Pid*.

Im Beispiel wurde für den Block inres kein Zustandsautomat festgelegt; damit ist dieser implizit vorhanden.

In einem hierarchisch strukturierten SDL-Programm werden unter Umständen die gleichen Namen für Definitionen mehrfach verwendet. Um Mehrdeutigkeiten zu vermeiden, gibt es Sichtbarkeitsregeln, die die Verdeckung eines Namens durch einen anderen festlegen. Um Bezug auf verdeckte Bezeichner zu nehmen, und verbleibende Mehrdeutigkeiten aufzulösen, kann jeder Name durch Angabe des Sichtbarkeitsbereichs qualifiziert werden. Dieser wird dazu in einem Paar von Kleiner- und Größerzeichen aufgeführt (<<*scope*>>)

3.4 Datentypen

Das Datentypsystem von SDL-92 basiert auf dem Kalkül der initialen Algebra [Z.100-92, Annex C]. Jeder Datentyp definiert eine Sorte. Diese Sorte wird durch eine Menge von Termen beschrieben. Zwischen den Termen wird mit Hilfe von Axiomen eine Äquivalenzrelation eingeführt (die *Termäquivalenz*). Die einzelnen Äquivalenzklassen sind dann die Werte des Datentyps.

Obwohl dieses Kalkül ausdrucksstark genug ist, um die in Programmiersprachen üblichen Datentypen zu beschreiben, weist es eine Reihe von Mängeln auf, die zur Ersetzung dieses Modells durch ein anderes in SDL-2000 führten:

1. Das Datentypsystem ist für Anwender schwer zu verstehen. Wird ein Anwender mit einem System von Operatoren und Axiomen konfrontiert, erschließt sich ihm oft nicht die Bedeutung des Datentyps.
2. Das System ist schwer zu implementieren. Selbst die einfache Frage der Feststellung einer Äquivalenz von Termen ist, im Allgemeinen, äquivalent zum Halteproblem [Sch02]. Wenngleich für viele Datentypdefinitionen eine effiziente Implementierung denkbar ist, wurde jedoch für SDL nie ein Werkzeug entwickelt, dass diese effiziente Implementierung findet.
3. Das Datentypsystem basiert auf einer Wertesemantik. Der Objektbegriff lässt sich mit diesem Kalkül nicht nachbilden. Beispielsweise initialisieren die folgenden Anweisungen zwei Variablen mit einem Strukturwert, wobei die Struktur zwei Felder besitzt:

   ```
   task a := (. 3, 4 .);
   task b := (. 3, 4 .);
   ```

   In diesem Beispiel ergibt sich aus der Definition der initialen Algebra zwingend, dass die Variablen a und b den gleichen Wert haben, da beide mit dem identisch selben Term belegt wurden. In einer objekt-orienterten Sprache, in der die Variablen a und b Objektreferenzen sind, erwartet man jedoch, dass nach diesen Zuweisungen die Variablen auf verschiedene Objekte verweisen; man versteht den Ausdruck auf der rechten Seite als Konstruktion eines Objekts. Da Objekte nicht nur durch Zustand und Verhalten, sondern auch durch ihre Identität beschrieben sind, ist der Begriff der Wertgleichheit allein unzureichend für ein objektorientiertes Datentypsystem.

Während der Entwicklung von SDL-2000 spielte die Rückwärtskompatibilität zu früheren Sprachversionen eine wesentliche Rolle: Auch wenn beim Einsatz von SDL-2000 Anpassungen alter Spezifikationen an neue Konstrukte nötig sind, so sollte doch in vielen Fällen eine einfache, möglichst automatisch durchführbare Anpassung möglich sein[6].

Aus diesem Grund wurde ein Datentypsystem, welches ausschließlich auf Objekten basiert (wie etwa das von Java) für SDL verworfen: Zu viele existierende Spezifikationen vertrauen auf die Wertesemantik; diese sollten möglichst ihre Semantik behalten.

Deshalb wurde für SDL zunächst ein zwei-teiliges Datentypsystem entworfen. Mit der Verallgemeinerung von Prozessen und Blöcken zu Agenten und der Einführung von Schnittstellen kam dann noch ein dritter Teil hinzu. Ein Datentyp in SDL ist entweder ein

- Wertetyp (*value type*),
- Objekttyp (*object type*) oder ein
- Pid-Typ (*Pid type*).

Wenn ein Wertetyp als Variablentyp oder Parametertyp verwendet wird, erfolgen Zuweisungen an diese Variablen oder Parameter als Wertzuweisungen. Der Wert der Variablen ist also nach der Zuweisung eine Kopie.

6. Der Autor dieser Arbeit hat in Appendix III von SDL-2000 ein Verfahren zur systematischen Konvertierung von SDL-92-Spezifikationen zu SDL-2000 vorgeschlagen.

Bei Variablen oder Parametern von Objekttypen erfolgen Zuweisungen per Referenz. Nach der Zuweisung verweist die Variable auf dasselbe Objekt wie der zugewiesene Ausdruck. Änderungen dieses Objekts über eine Referenz wirken sich dann auch auf alle anderen Referenzen auf das gleiche Objekt aus.

Eine Abweichung von dieser Referenzsemantik findet sich bei Austausch von Signalen zwischen Agenten. Hier wird beim Versenden des Signals eine Kopie aller versendeten Signalparameter erzeugt, sofern sich der Empfänger nicht im gleichen Prozess wie der Sender befindet. Damit können Agenten in verschiedenen Prozessen nicht die gleichen Objekte referenzieren. Diese Abweichung erlaubt effiziente Implementierungen von Objekttypen, da bei Zugriff auf ein Objekt stets davon ausgegangen werden kann, dass sich dieses Objekt lokal in dem Prozess befindet.

Mit der Einführung von Objekttypen einher ging die Aufnahme von polymorphen Variablen und von Methoden. Methoden erlauben die Kapselung des Objektzustands: Eine Methode stellt die Programmierschnittstelle eines Objekts nach außen zur Verfügung. Der Objektzustand kann damit zum von außen nicht zugänglichen Implementierungsdetail deklariert und lediglich innerhalb von Methoden manipuliert werden.

Für Wertetypen muss der Typ des Ausdrucks in einer Zuweisung gleich dem Typ der Variablen sein. Für Objekttypen kann der Typ des Ausdrucks auch eine Spezialisierung des Variablentyps sein. In diesem Fall unterscheidet sich nach der Zuweisung der statische Typ der Variablen vom dynamischen Typ des referenzierten Objekts.

Unter Umständen ist es erforderlich, bei einer polymorphen Variable auf Eigenschaften der Spezialisierung zuzugreifen. Dies ist in SDL durch zwei Konstrukte möglich, nämlich:

1. Virtuelle Methoden: Bei einer virtuellen Methode wird die Methodendefinition auf Grund des dynamischen Typs der Methodenparameter ausgewählt. Die Methoden werden als „spät“ gebunden – die zu rufende Methode kann erst im Moment des Rufs bestimmt werden.
2. Zuweisungsversuch (*assignment attempt*): Dieses in C++ als „*type cast*“ bekannte Konstrukt erlaubt die Umwandlung einer Referenz auf einen Basistyp in eine Referenz auf eine Spezialisierung. Anstatt dafür ein neues Syntaxkonstrukt einzuführen, wurde die Semantik von Zuweisungen erweitert: Ist der Typ des Ausdrucks in einer Zuweisung Basistyp des Typs der Variablen, so ist diese Zuweisung nach der statischen Semantik richtig. Dynamisch wird überprüft, ob der aktuelle Typ des Ausdrucks eine Spezialisierung des Variablentyps ist. In diesem Fall ist die Zuweisung erfolgreich. Anderenfalls wird der Wert null an die Variable zugewiesen; der Zuweisungsversuch ist dann gescheitert.

Beispiel 8. Diese Konstellationen sollen an einem Beispiel erläutert werden. Gegeben seien drei Typen

```
object type T1{
    literals A;
}
object type T2 inherits T1{
    literals B;
}
object type T3 inherits T2{
    literals C;
}
```

Da bei einer Spezialisierung eines Typs die Spezialisierung alle Eigenschaften des Basistyps erbt, enthält der Typ T1 das Literal A, der Typ T2 die Literale A und B und der Typ T3 die Literale A, B und C. Damit werden nun folgende Variablendefinitionen und Zuweisungen möglich:

```
dcl t1 T1, t2 T2;
task t1 := C;     /* Polymorphe Zuweisung von <<type T3>> C */
task t2 := t1;    /* Zuweisungsversuch, der dynamische Typ ist T3: t2 wird <<type T3>> C */
task t1 := A;    /* Zuweisung von <<type T1>> A*/
task t2 := t1;   /* Zuweisungversuch; der dynamische Typ ist T1: t2 wird null */
```

Neben Objekt- und Werttypen gibt es in SDL Pid-Typen. Eine Pid-Variable enthält eine Referenz auf einen Agenten (oder den Wert null, wenn kein Agent referenziert ist, insbesondere als Initialwert einer Pid-Variablen). Pid-Variablen müssen nicht unbedingt mit dem Typ Pid definiert werden, der Referenzen auf beliebige Agenten repräsentiert. Statt dessen können auch Schnittstellen als Datentypen verwendet werden – jede Schnittstellendefinition impliziert einen Pid-Typ. In einer Agentendefinition kann definiert werden, dass ein Agent A eine bestimmte Schnittstelle S unterstützt. Pid-Variablen des Typs S können dann Refererenzen auf den Agenten A aufnehmen.

3.4.1 Vordefinierte Datentypen

Eine Reihe von Datentypen ist in SDL vordefiniert. Alle dieser Typen sind Wertetypen, darunter[7]

- der Typ Boolean,
- die Zahlentypen (Integer, Real),
- Typen für Zeichen und Zeichenketten (Character, Charstring) und
- Typen für Zeit und Zeitspannen (Time, Duration).

Eine Reihe von Typen sind mit Kontextparametern behaftet, um sie als Containertypen für andere Datentypen einsetzen zu können, darunter die Typen

- String (zur Repräsentation einer Folge von Werten eines Typs),
- Array (zur Indizierung eines Typs mit einem anderen),
- Vector (zur Indizierung eines Typs durch Elemente eines Intervalls ganzer Zahlen) und
- Powerset (zur Repräsentation von Mengen aus Elementen eines Typs).

3.4.2 Nutzerdefinierte Datentypen

Auf Basis der vordefinierten Datentypen können anwendungsspezifische neue Datentypen definiert werden. Dazu gibt es drei Typkonstruktoren:

- Literaltypen, deren mögliche Werte durch Aufzählung aller Elemente angegeben werden,
- Strukturtypen, die durch Bildung des Kreuzprodukts aus bestehenden Typen entstehen und
- Choice-Typen, die durch Vereinigung bestehender Typen entstehen.

Die genauen syntaktischen Regeln für jeden dieser Typkonstruktoren werden in Abschnitt 6.5 vorgestellt.

3.4.3 Auflösung von Operatornamen

In SDL werden in der Regel Bezeichner mit ihrer Definition auf Grund hierarchischer Sichtbarkeitsregeln aufgelöst: Ein Name eines Signals in einem inneren Agenten verdeckt ein gleichnamiges Signal eines äußeren Agenten. Wenn eine Definition verdeckt ist, kann auf sie durch Qualifizierung zugegriffen werden. Außerhalb ihres Sichtbarkeitsbereichs kann eine Definition nicht referenziert werden.

Obige Sichtbarkeitsregeln funktionieren bei Datentypen aus folgenden Gründen nicht:

- Die Namen von Literalen und Operatoren sind innerhalb des Datentyps deklariert, sind also ohne weiteres an der Stelle ihrer Verwendung gar nicht sichtbar.

7. Die vollständige Liste aller Datentypen findet man in Annex D von [Z.100-00].

- Viele Operatoren haben gleiche Namen (beispielsweise der Operator „+“), so dass eine Operatorverwendung oft mehrdeutig ist.

Beispielsweise ist es nicht möglich, für den Ausdruck „2+4“ einen Typ anzugeben, da die Literale vom Typ Integer, Real oder Duration sein können; der Additionsoperator hat dann einen entsprechenden Typ.

Aus diesen Gründen wird in SDL ein Verfahren der kontextabhängigen Auflösung von Operatornamen und Literalnamen verwendet (*resolution by context*).

Dieses Verfahren besteht aus folgenden Elementen:

- Ist ein Datentyp an einer Stelle der Spezifikation sichtbar, so sind auch alle seine Literale und Operationen dort sichtbar.
- Für eine Aktion (Zuweisung, Timer-Start, Signalversendung) werden alle in dieser Aktion auftretenden Operatoren ermittelt, und für jeden dieser Operatoren alle sichtbaren Definitionen.
- Für jede dieser Definitionen wird überprüft, ob sie typrichtig ist. Falls alle Definitionen typfalsch sind, ist der Ausdruck falsch.
- Unter den typrichtigen Operatoren werden diejenigen ausgewählt, die die kleinste Zahl von polymorphen Typverwendungen aufweisen.
- Gibt es mehrere solcher Operatoren, ist der Ausdruck mehrdeutig und damit falsch; gibt es genau eine Interpretation, ist der Ausdruck richtig.

Betrachtet man beispielsweise die Definitionen

```
dcl a Integer;
task a := a + 1;
```

so wird für den Bezeichner a zunächst festgestellt, dass es sich um eine Variable vom Typ Integer handelt. Das Literal 1 kann die Typen Integer, Real oder Duration haben. Für den Operator gibt es folgende Interpretationen:

```
"+" ( Integer, Integer) -> Integer;
"+" ( Real, Real) -> Real;
"+" ( Duration, Duration) -> Duration;
"+" ( Time, Duration) ->Time;
"+" ( Duration, Time) ->Time;
```

Von diesen Interpretationen erlaubt nur die Version

```
task a := <<type Integer>>"+" (a, <<type Integer>> 1);
```

eine typrichtige Interpretation. Diese ergibt dann auch die Semantik des Ausdrucks.

Wie bereits erwähnt, kann durch die statische Auflösung einer Operation nicht immer ermittelt werden, welche Operation gerufen wird: Ergibt die kontextabhängige Auflösung eine virtuelle Methode, so erfolgt die tatsächliche Bindung einer konkreten Methodendefinition erst während der Abarbeitung des Ausdrucks.

3.5 Typen und Objektorientierung

Seit SDL-92 muss die Struktur des Systems nicht mehr unbedingt aus einer festen Hierarchie von Blöcken und Prozessen bestehen. Statt dessen ist es auch möglich, Typen von Systemen, Blöcken und Prozessen zu bilden und diese flexibel in verschiedenen Systemen zusammenzusetzen. Typen können unter anderem in folgenden Fällen eingesetzt werden:

1. Durch Definition von Block- und Prozesstypen innerhalb von Paketen (**package**) kann die gleiche Teilhierarchie in mehreren Systemen verwendet werden.
2. Durch Vererbung (Spezialisierung) von Typen kann ein vorhandener Typ um weitere Eigen-

schaften erweitert werden.

3. Durch Definition virtueller Typen können Typen spät gebunden werden: Der Typ, der für eine Instanzierung verwendet wird, hängt dann von der Spezialisierung des Typen ab, in dem er verwendet wurde.
4. Durch Definition von Kontextparametern kann ein Typ an verschiedene Kontexte angepasst werden. Der so entstehende Typ ist eine abstrakte Typschablone. Von diesem Typ können nicht unmittelbar Instanzen gebildet werden. Statt dessen muss durch die Angabe von aktuellen Kontextparametern aus der Typschablone ein Typ gebildet werden.

3.5.1 Typisierung

Die Definition eines Typs (Systemtyp, Blocktyp, Prozesstyp) erfolgt ähnlich zur Definition eines Typs der entsprechenden nicht-typorientierten Struktur. Abbildung 4 definiert einen Block, in dem der Prozess Responder auf Basis des Prozesstyps ResponderTyp definiert wurde.

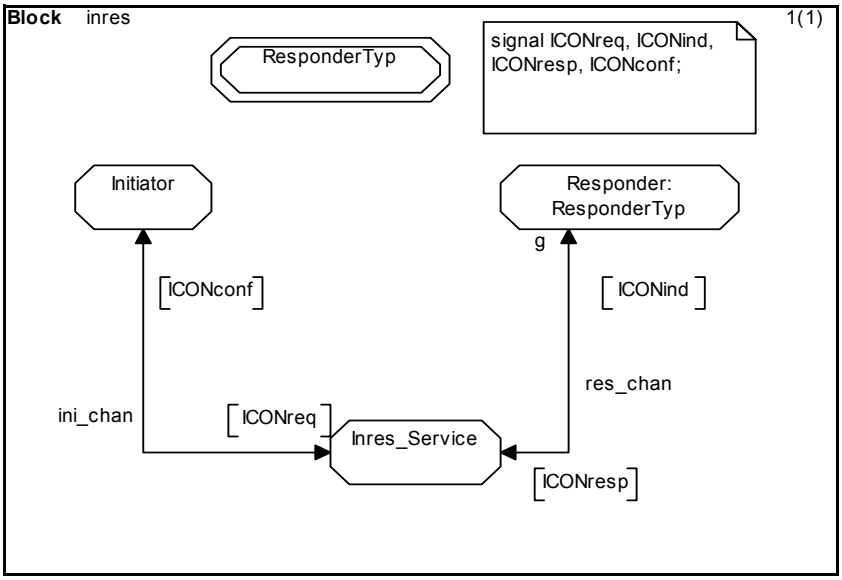

Abbildung 4: Verwendung von Typen

Durch die Definition eines Prozesstyps können mehrere Prozesse definiert werden, die alle das gleiche Verhalten besitzen[8]. Zur Definition mehrerer Systeme auf Grundlage der gleichen Typdefinitionen wurden in SDL-92 Pakete (*packages*) eingeführt. In Paketen können folgende Konstrukte definiert werden:

- Agententypen (System-, Block-, Prozesstypen),
- Signale,
- Signallisten,
- entfernte Prozeduren und Variablen,
- Datentypen,
- Prozeduren und
- Makrodefinitionen (grafische Makros sind in SDL-2000 nicht mehr unterstützt).

8. Streng genommen beschreibt auch schon in SDL-88 eine Prozessdefinition mehrere Prozessinstanzen, da der Prozess eine Prozessinstanzmenge definiert. Jedoch befinden sich all diese Instanzen immer im gleichen Block; Prozesstypen erlauben die Definition gleicher Prozesse in verschiedenen Blöcken oder Systemen.

In SDL-2000 dürfen Pakete außerdem
- Zustandstypen (*state types*),
- Schnittstellendefinitionen (*interfaces*) und
- Ausnahmedefinitionen (*exceptions*)

enthalten.

In SDL-92 ist die Semantik von Typen durch ein Ersetzungsmodell definiert: Mit jeder Verwendung eines Typs wurde eine Kopie aller Eigenschaften des Typs erzeugt, und die Verwendung des Typs durch das entsprechende nicht-typorientierte Konstrukt ersetzt. In SDL-2000 ist die Semantik von Typen „direkt“ erklärt. Falls eine Spezifikation nicht-typorientierte Konstrukte enthält, werden (durch ein Ersetzungsmodell) anonyme Typen eingeführt und diese Konstrukte durch typbasierte Konstrukte ersetzt.

3.5.2 Spezialisierung

Die Behandlung von Vererbung (im SDL-Zusammenhang oft Spezialisierung genannt) erfolgte in SDL-92 für alle relevanten Konzepte im Wesentlichen einheitlich. Vererbung ist in SDL-92 möglich für
- Systemtypen,
- Blocktypen,
- Prozesstypen,
- Servicetypen (das Service-Konzept wurde in SDL-2000 gestrichen),
- Prozeduren,
- entfernte Prozeduren,
- Datentypen und
- Signale.

In SDL-2000 ist Vererbung zusätzlich erlaubt für
- Zustandstypen und
- Schnittstellen.

SDL unterstützt für fast alle dieser Konzepte nur Einfachvererbung. Lediglich für Schnittstellen ist Mehrfachvererbung erlaubt. Bei der Definition eines Typs als Spezialisierung eines anderen erbt der spezialisierte Typ alle Eigenschaften des Basistyps und kann diesen weitere Eigenschaften hinzufügen.

Die Definition eines Typen als Spezialisierung eines anderen Typen bedeutet in SDL-92 noch nicht, dass diese Typen polymorph verwendet werden können, im Sinne des Liskovschen Substitutionsprinzips [Lis87]. Für die meisten Konzepte ist die Polymorphie auch nicht relevant. So war es in SDL-92 nicht vorgesehen (und auch nicht sinnvoll), Referenzen auf Blöcke zu erwerben, so dass die Frage der Polymorphie von Blöcken sich auch gar nicht stellt. Zur Referenzierung von Prozessen war nur ein einziger Datentyp (Pid) vorgesehen, der polymorph Referenzen auf Prozesse beliebiger Typen repräsentieren konnte. Auch die Polymorphie von Prozeduren ist nicht interessant, weil sich auch Prozeduren nicht referenzieren lassen.

Vor allem für Datentypen stellte sich in der Praxis das Fehlen von Polymorphie als Problem dar [Sch02]. Dieser Mangel wurde in SDL-2000 durch das objekt-orientierte Datentypkalkül behoben (siehe Abschnitt 3.4).

Zusätzlich wurde in SDL-2000 Polymorphie von Agentenreferenzen eingeführt, die auf der Basis von Schnittstellen definiert wird. Eine Schnittstelle umfasst dabei eine Menge von Signalen, die ein Agent konsumieren kann. Bei der Definition des Agenten wird deklariert, welche Schnittstellen er unterstützt. Eine Variable, deren Typ eine Schnittstelle ist, kann polymorph Referenzen auf alle Agenten aufnehmen, die diese Schnittstelle unterstützen.

3.5.3 Virtuelle Typen und virtuelles Verhalten

Um einen Agententyp als Basistyp verwenden zu können, muss sich dieser Typ durch Erweiterung an die geforderte Anwendung anpassen lassen. Dazu reicht es oft nicht, zu dem Typ lediglich weitere Eigenschaften (neue enthaltene Agenten, neue Zustandsvariablen, neue Zustände) hinzuzufügen: Das bereits bestehende Verhalten des Basistyps muss auch geändert werden, um von diesen erweiterten Eigenschaften Gebrauch zu machen.

Zu diesem Zweck erlaubt SDL die Definition virtueller Konstrukte. Wird ein Typ, der virtuelle Konstrukte enthält, spezialisiert, so können die virtuellen Konstrukte durch neue redefiniert werden.

Dabei gibt es zwei Arten virtueller Konstrukte: Virtuelle Verhaltenskonstrukte und virtuelle Typen.

Virtuelle Typen können im Basistyp zur Definition der Struktur verwendet werden; so kann in einem Blocktyp ein virtueller Prozesstyp definiert und instanziert werden. In der Spezialisierung des Basistyps kann der virtuelle Prozesstyp redefiniert werden. Alle Instanzierungen des virtuellen Typs verwenden dann die Redefinition. In diesem Zusammenhang gelten auch virtuelle Prozeduren als Typen – die Rufe der Prozedur sind ihre „Instanzierungen“.

Virtuelle Verhaltenskonstrukte wird durch die Definition virtueller Transitionen und virtueller Methoden definiert; letztere werden in Abschnitt 3.4 vorgestellt.

Virtuelle Transitionen erlauben es, in der Spezialisierung eines Agententyps die Reaktion des Agenten auf den Empfang eines Signals zu ersetzen; die Transition wird dann redefiniert. Als Spezialfall kann sogar die Starttransition als virtuell definiert werden. Der spezialisierte Agent führt dann die Starttransition der Spezialisierung aus.

Eine Redefinition eines virtuellen Typs oder einer virtuellen Transition kann als **final** deklariert werden. Dieser Typ (oder diese Transition) kann dann in weiteren Spezialisierungen des Agententyps nicht mehr redefiniert werden.

3.5.4 Kontextparameter

Um einen Typ möglichst flexibel einzusetzen, ist es wünschenswert, von Details mancher Eigenschaften des Typs zu abstrahieren und den Typ so allgemein wie möglich zu formulieren. SDL bietet hierzu das Konzept parametrisierter Typen, die durch Kontextparameter spezialisiert werden. Abbildung 5 definiert einen Prozesstyp, der die Registrierung von Daten realisiert. Um welche Daten es sich dabei handelt, ist für die Logik nicht von Bedeutung. Dies wird durch den nicht weiter eingeschränkten Kontextparameter t definiert. Festgelegt wird lediglich, dass die Registrierung über ein Signal register erfolgt und der Test, ob ein Wert registriert wurde, über das Signal lookup. Auch die Namen der Signale sind hierbei Kontextparameter – instanziert man den Typ, können andere Signalnamen festgelegt werden.

Bei der Definition von Kontextparametern können Bedingungen (constraints) festgelegt werden, die für die Kontextparameter gewisse Eigenschaften fordern, beispielsweise, dass ein Agententenkontextparameter mindestens eine spezifizierte Schnittstelle einhält. Bei Spezialisierung des parametrisierten Typs müssen die aktuellen Parameter diese Bedingungen einhalten.

3.6 Konkrete Syntax

SDL wird in zwei verschiedenen Formen notiert: einer grafischen (SDL/GR) und einer textuellen (SDL/PR). Die grafische Notation ist zur Bearbeitung durch den Systemdesigner vorgesehen. Die textuelle Notation zur Bearbeitung durch Programme und zum Austausch zwischen Programmen.

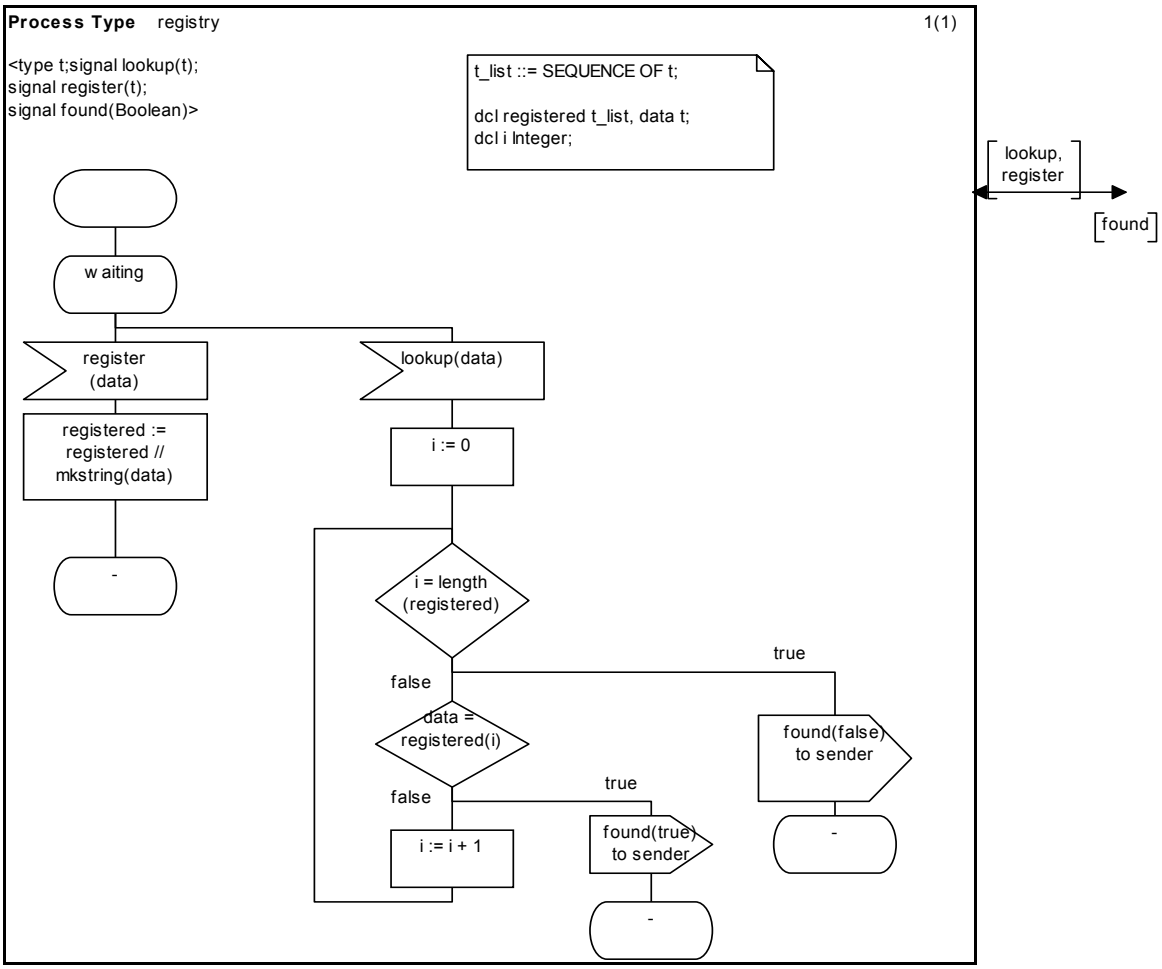

Abbildung 5: Kontextparameter

Für manche Konstrukte ist in SDL keine eigene grafische Form vorgesehen, beispielsweise für Datendefinitionen. An diesen Stellen wird dann auch in der grafischen Notation Text verwendet, beispielsweise innerhalb des Textsymbols (⬚)

In der Sprachversion SDL-2000 wurde SDL in seinen syntaktischen Regeln an die Sprachfamilie C/C++/Java angepasst. Dazu wurden folgende Änderungen eingeführt:

- Der Unterschied zwischen Groß- und Kleinschreibung ist nun signifikant. Schlüsselwörter gibt es in zwei Formen: einer, die aus Großbuchstaben und einer, die aus Kleinbuchstaben besteht.
- Zur Gruppierung von Definitionen, insbesondere für Datentypen, dürfen nun geschweifte Klammern verwendet werden.
- Zur Abtrennung von Feldnamen bei strukturierten Datentypen darf nun der Punkt (.), zur Indizierung eckige Klammern ([und]) verwendet werden.
- Da die grafische Formulierung von komplizierten Algorithmen recht aufwändig ist, ist es nun möglich, Algorithmen auch in einer textuellen Form zu notieren. Diese textuelle Form kann mit der grafischen Form gemischt werden.

Aus Rückwärtskompatibilität zu früheren Sprachversionen können die alten Konstrukte weiter verwendet werden (beispielsweise die Verwendung des Ausrufezeichens zur Abtrennung von Feldnamen).

Im folgenden wird für Beispiele in der Regel die textuelle Notation verwendet. Nach Erfahrung des Autors ist die grafische Syntax besonders geeignet, um die Struktur eines Agenten

schnell zu erfassen. Die textuelle Form ist besser zur Darstellung von Algorithmen und Datenstrukturen geeignet.

3.7 Weitere Konzepte

Eine Reihe von Konzepten sind zwar zentral für die Sprache SDL, im Rahmen dieser Arbeit aber von untergeordneter Bedeutung. Dazu zählt vor allem die Kommunikation zwischen Agenten:

- Signale werden grundsätzlich über *Kanäle* ausgetauscht. Ein Kanal verbindet zwei Agenten, die Verbindung kann unidirektional oder bidirektional sein.
- Kanäle enden an *Gates* des Agenten. Ein Gate legt fest, welche Signale den Agenten verlassen, und welche vom Agenten empfangen werden können.
- Zur Kommunikation zwischen Agenten können nicht nur Signale, sondern auch entfernte Prozeduren (*remote procedure*) und entfernte Variablen (*remote variable*) verwendet werden.

Weiterhin bietet SDL eine Reihe von Konstrukten, mit denen Nicht-Determinismus formuliert werden kann:

- Ein Agent kann jederzeit eine *spontane Transition* ausführen, ohne einen externen Stimulus zu empfangen.
- Über das Konstrukt der *nicht-deterministischen Decision* kann aus einer Reihe von Aktionen eine willkürlich ausgewählt werden.

3.8 Fazit

Mit der Definition von SDL-2000 wurde die Sprache auf harmonische Weise weiterentwickelt. Dieses Kapitel bietet nur einen Überblick über die Sprache SDL, um eine systematische Einführung zu erhalten und komplexe Anwendungsbeispiele zu studieren, sollte der Leser auf geeignete Literatur zurückgreifen, etwa [OFM+94] und [Dol01].

4 Der Entwicklungsprozess der formalen SDL-Semantikdefinition

Die Sprachspezifikation von SDL-2000 besteht aus mehreren Teilen, von denen manche vorwiegend informal in englischer Sprache und andere vorwiegend mit formalen Sprachen notiert sind:

- Der Haupttext der Sprachdefinition in der ITU-Empfehlung Z.100 [Z.100-00] erklärt die Syntax von SDL durch Angabe der EBNF und die Semantik durch englischen Text.
- Annex A ist der Index der nicht-terminalen Symbole der Grammatik.
- Annex D definiert die vordefinierten Daten (package predefined) unter Verwendung einer formalen Syntax. Die Semantik dieser Datentypen wird dort informal angegeben.
- Annex F [Z.100F] definiert die formale Semantik; dieser Annex gliedert sich in Teil F.1 (Übersicht), Teil F.2 (statische Semantik) und Teil F.3 (dynamische Semantik).

Diese Dokumente wurden über einen Zeitraum von mehreren Jahren in verschiedenen Arbeitsgruppen entwickelt. Dabei entstanden die informale und die formale Sprachdefinition parallel zueinander. Um trotzdem zu einer konsistenten Sprachdefinition zu gelangen, wurde ein Entwicklungsprozess eingesetzt, bei dem systematisch die fertig gestellten Teile der informalen Sprachdefinition formalisiert wurden.

In diesem Kapitel werden typische Probleme dieses Entwicklungsprozesses vorgestellt. Sowohl die Entdeckung dieser Probleme als auch ihre Behebung erfordern einen integrierten Entwicklungsprozess, in dem die formale Semantikdefinition möglichst ständig der informalen Sprachdefinition entspricht. Nur so konnten eventuell nötige Änderungen noch vor Verabschiedung der Sprachdefinition entdeckt und ausgeführt werden.

Durch die beteiligten Arbeitsgruppen sowie die Dokumente, die bei der Entwicklung entstanden, gliederte sich der Entwicklungsprozess in verschiedene Phasen, so wie sie in Abbildung 6 dargestellt sind.

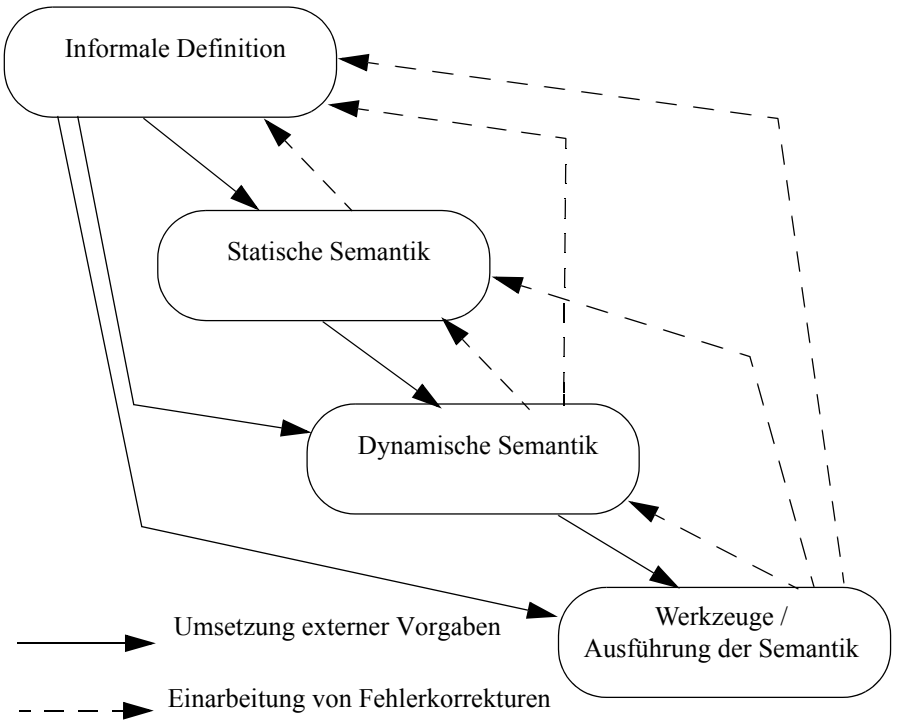

Abbildung 6: Entwicklungsprozess der formalen Semantikdefinition

In der Praxis erfordert diese Arbeitsweise Kompromisse, da es natürlich nicht möglich ist, Änderungen der informalen Sprachdefinition sofort in die formale Semantikdefinition einfließen zu lassen. Als Folge davon sind auch in der verabschiedeten Sprachdefinition Fehler enthalten, die nun im Prozess von Revisionen entfernt werden müssen. Auch hierfür werden im folgenden einige Beispiele gegeben.

4.1 Übersicht über die Werkzeugkette

Inkonsistenzen und andere Problemen der Sprachdefinition wurden durch zwei Vorgehensweisen aufgedeckt. Zum einen führt allein der Versuch, ein bestimmtes Konzept zu formalisieren, zu der Beobachtung, dass bestimmte Konstellationen keine klare Bedeutung haben. In diesem Fall führt also das bloße „Draufschauen“ zur Aufdeckung von Fehlern. Obwohl dieses Verfahren wenig systematisch und zuverlässig erscheint, ist es doch bei sorgfältiger Durchführung recht wirksam: Wenn beispielsweise bei der Formulierung eines Algorithmus festgestellt wird, dass dieser Algorithmus nicht über die nötigen Daten verfügt, wird der Sprachentwickler versuchen, dieses Problem durch Erweiterung des formalen Modells um diese Daten zu beheben. Dabei stellt er unter Umständen fest, dass diese Erweiterung gar nicht möglich ist und der Algorithmus revidiert werden muss.

Zum anderen wurden zahlreiche Probleme durch den Einsatz von Werkzeugen entdeckt. Die verwendeten Werkzeuge werden im Kapitel 9 vorgestellt. An dieser Stelle soll ein Überblick über die Werkzeuge genügen, um darzustellen, welche Arten von Problemen durch sie aufgedeckt werden können. Ein grober schematischer Ablauf der Werkzeugkette ist in Abbildung 7 dargestellt.

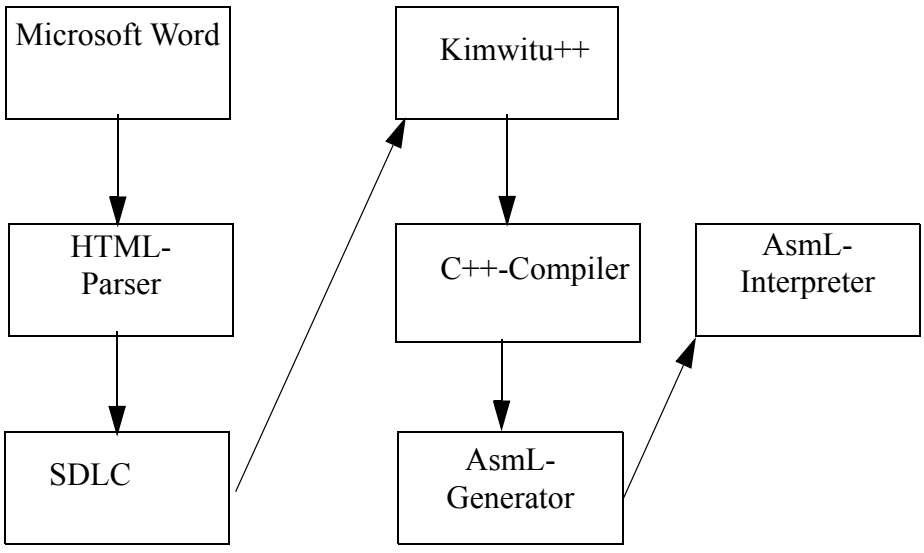

Abbildung 7: Verwendete Werkzeuge

Aufgrund der Vorgabe durch die ITU-Verwaltung musste die formale Semantik mit dem Werkzeug Microsoft Word [Mic99] erstellt werden. Zur Formulierung dieser Semantik wurde dabei von folgenden Eigenschaften des Werkzeugs Gebrauch gemacht:

- Durch Festlegung von Textmarken und Hypertext-Querverweisen ist eine einfache Navigation im Dokument möglich. Beispielsweise verweist jeder Aufruf einer Funktion auf die

Definition der Funktion.

- Durch Auto-Korrektur-Einträge wird die Erzeugung dieser Querverweise vereinfacht. Beispielsweise führt die Eingabe der Zeichenfolge \gdecision zur Ersetzung durch einen Querverweis auf die Grammatikregel *Decision*.
- Durch Visual-Basic-Macros wird die Definition dieser Auto-Korrektur-Einträge vereinfacht. Ein solches Makro untersucht das Dokument nach Grammatikregeln, Funktionsdefinitionen und anderen Definitionen und fügt die erforderlichen Autokorrektureinträge ein.
- Durch die Verwendung von Absatzformaten und Zeichenformaten wird einerseits die Generierung des Autokorrektur-Generators gesteuert, andererseits die Extraktion der formalen Teile für die weitere Verarbeitung.

Im Word-Dokument wechseln formale und informale Teile einander ab. Die informalen Teile erläutern die erwünschte Funktionsweise der darauf folgenden Formeln. Zur Verarbeitung der formalen Sprachdefinition sind die informalen Teile nicht erforderlich, und die formalen Teile lassen sich nicht einfach verarbeiten, solange sie im Microsoft-Word-Format gespeichert sind. Deshalb schließt sich der Erstellung des Word-Dokuments ein Extraktionsschritt an, der die formalen Teile extrahiert und in einer ASCII-basierten Sprache abspeichert.

Dieses Extraktionsverfahren verwendete ursprünglich ebenfalls Visual-Basic-Makros. Leider zeigte sich, dass die genaue Funktionsweise dieser Makros von der verwendeten Word-Version abhängt, so dass sich eine Pflege der Makros als schwierig erwies. Aus diesem Grund wurde der Extraktionsprozess auf Basis der HTML-Fähigkeiten von Word reimplementiert: Das Word-Dokument wird als HTML-Dokument abgespeichert. Ein HTML-Parser verarbeitet dann dieses Dokument und generiert die extrahierten Dateien, die nur reinen ASCII-Text enthalten.

Diese Dateien werden vom Compiler SDLC (der während des Projekts entstand) in Kimwitu++-Code übersetzt, welcher dann von Kimwitu++ [vLP02] in C++-Code und vom C++-Compiler in ein ausführbares Programm übersetzt wird.

Dieses Programm ist ein aus der formalen Semantik automatisch abgeleiteter Referenzcompiler, der aus der Eingabe einer SDL-Spezifikation eine Reihe von AsmL-Fragmenten[9] generiert. Diese werden zusammen mit weiteren AsmL-Fragmenten (die aus dem Word-Dokument stammen) vom AsmL-Compiler [Mic02] in ein C#-Programm übersetzt, welches dann vom C#-Compiler [Arc01] in ein Microsoft-.NET-Programm übertragen wird.

Die Abarbeitung dieses .NET-Programms spiegelt dann einen gültigen Ablauf des SDL-Programms wider.

Nahezu jeder dieser Schritte kann scheitern. Im folgenden werden typische beobachtete Fehlerfälle und deren Lösung beschrieben.

4.2 Fehler bei der SDLC-Kompilierung

Wenn der SDLC-Compiler Fehler liefert, sind das in der Regel Fehler syntaktischer Art in der extrahierten Eingabe. Diese Fehler können allerdings mehrere Ursachen haben.

4.2.1 Fehler im Extraktionswerkzeug

Bei der Extraktion der von Microsoft Word generierten HTML-Datei sind verschiedene Besonderheiten von Word zu beachten, die immer wieder zu Fehlern im Extraktor geführt haben:

- Leerzeichen
  Die Verwendung von Leerzeichen im generierten HTML bedeutet nicht immer, dass auch im Word-Dokument Leerzeichen vorhanden waren. An manchen Stellen ist es erforderlich,

9. AsmL [Mic02] ist ein Werkzeug für die Ausführung von Abstract State Machines (ASM); diese werden in Abschnitt 5.3.1 erläutert.

in den extrahierten Dateien Leerzeichen einzufügen, die im HTML nicht erwähnt sind; an anderen Stellen müssen Leerzeichen weggelassen werden.

- Mathematische Sonderzeichen
  In den Formeln der Semantikdefinition wird von mathematischen Symbolen (etwa ∃ und ∀) Gebrauch gemacht. Diese werden im HTML-Dokument in der Schriftart Symbol repräsentiert. Im extrahierten Dokument ist statt dessen eine ASCII-Repräsentation (etwa \exists und \forall) erforderlich. Wird im Worddokument ein weiteres Zeichen verwendet, welches vom Extraktor nicht unterstützt ist, muss entweder der Extraktor verändert oder auf die Verwendung dieses Zeichens verzichtet werden.
- Interpunktionszeichen
  In der Definition von Funktionen der formalen Semantik wurde vom mathematischen Stil Gebrauch gemacht, durch Apostroph-Zeichen ähnliche Variablen zu bezeichnen, etwa in dem Prädikat

  $$\forall d, d' \in \mathit{ENTITYDEFINITION}_1 : d.\mathit{entityKind}_1 = d'.\mathit{entityKind}_1 \wedge d \neq d' \Rightarrow d.\mathit{identifier}_1 \neq d'.\mathit{identifier}_1$$

  Der Compiler SDLC akzeptiert dieses Zeichen nicht in Bezeichnern und hat also derartige Konstruktionen abgelehnt. Die Ursache dieses Problems ist die ungenaue Syntaxdefinition der Formelnotation in Annex F, die nicht festlegt, ob Apostrophe verwendet werden dürfen oder nicht. Zur Lösung gab es mehrere Alternativen: Die formale Semantikdefinition hätte geändert werden können, so dass sie solche Bezeichner nicht mehr verwendet. Das Extraktionswerkzeug hätte erweitert werden können, so dass es transparent diese Bezeichner umformuliert. Schließlich wurde SDLC erweitert, so dass der Compiler diese Apostroph-Zeichen bei der Generierung von kimwitu++ in die Zeichenkette PRIME überführt. Die Umsetzung dieser Zeichen in SDLC erlaubt es, den Originaltext über möglichst viele Verarbeitungsschritte unverändert zu lassen.

4.2.2 Formatierungsfehler im Word-Dokument

Der Compiler SDLC unterscheidet verschiedene Namensräume für Funktionen, Domänen, ASM-Regeln und ASM-Variablen. Diese werden im Word-Dokument durch Zeichenformate und in der extrahierten SDLC-Eingabe durch Präfixe (f-, d-, r- und a-) repräsentiert. Der Extraktor gibt diesen Präfix immer dann aus, wenn im HTML-Dokument ein -Element mit dem entsprechenden Zeichenformat auftritt. Dabei traten folgende Probleme auf:

- Falsche Textauszeichnung
  Die Formatierung des Bezeichners erstreckt sich nicht nur auf den Text des Bezeichners, sondern auch auf Leerzeichen vor und nach dem Bezeichner. Wählt man beispielsweise in Word ein Wort durch Doppelklick aus, so wird das Leerzeichen dahinter mitausgewählt. Ändert man dann das Zeichenformat des selektierten Bereichs, wird auch dieses Leerzeichen mitformatiert. Endet eine Zeile in einem Zeichenformat, so setzt sich dieses Zeichenformat bei Einfügen einer Zeile in dieser Folgezeile fort. In der Regel sind dann dort auch Leerzeichen formatiert. Diese formatierten Leerzeichen führen dann dazu, dass im extrahierten Text zwischen dem Präfix und dem präfigierten Bezeichner Leerzeichen zu finden sind, die von SDLC abgelehnt werden. In diesen Fällen musste die fehlerhafte Formatierung in Word beseitigt werden.
- Fehlende Textauszeichnung
  Manche Bezeichner waren nicht entsprechend formatiert, insbesondere, wenn die Eingabe der Formatierung in Word umständlich ist. Für Funktionen und Domänen ist die Eingabe der richtigen Formatierung ein Seiteneffekt der Auto-Korrektur-Einträge. Für ASM-Variablennamen gibt es keine solchen Auto-Korrektur-Einträge, weil der gleiche Variablenname unter Umständen in vielen Zusammenhängen verwendet wird, und sich demzufolge keine einzelne Definition als Ziel des Querverweises festlegen lässt. In diesen Fällen war es erfor-

derlich, die fehlende Formatierung nachträglich einzugeben.

4.2.3 Syntaxfehler

Bei der ersten Verarbeitung der formalen Semantikdefinition durch die Werkzeugkette sind zahlreiche Syntaxfehler gemeldet worden, die sich nicht auf falsche Formatierung zurückführen lassen. Diese Fehler lassen sich wie folgt kategorisieren (siehe auch Abschnitt 9.7):

- Systematische Verwendung nicht erlaubter Syntax
  Eine gewisse Menge systematischer Fehler entstand dadurch, dass den Autoren der formalen Semantik die konkreten Syntaxregeln für die Formeln nicht in allen Fällen geläufig waren. Sie haben diese Regeln teilweise aus dem Gedächtnis angewandt und teilweise durch Abschreiben von Formeln, die unter Umständen ihrerseits falsch waren. Ein typisches Beispiel für dieses Problem ist die Syntax das **case**-Konstrukts, welches richtig wie folgt zu verwenden ist:

  case *d'* **of**
  | *TypeDefinition*$_1$*Interface-definition* => *d* = *getEntityDefinition*$_1$(*d'*.**s-***Identifier*, *d'.entityKind*$_1$)
  | *Interface-definition* =>
  dInterface-definition
  (*dataIdData-type-identifier*: *dataId.parentAS1= d' d* = *getEntityDefinition*$_1$(*dataId*, **sort**))
  | *Syntype-definition* =>
  isDirectSuperType$_1$(*d*,*d'.derivedDataType*$_1$)
  otherwise *False*
  endcase

  In Anlehnung an andere Sprachen wurde von dieser Syntax aber versehentlich abgewichen: Statt des Schlüsselworts **of** wurde das Schlüsselwort **in** verwendet, und statt des Trennzeichens **=>** (welches das case-Muster von Ergebnisausdruck trennt) wurde das Trennzeichen **:** verwendet. Diese Fehler wurden in der Regel erst vom SDLC-Parser gefunden, und mussten im Word-Dokument korrigiert werden.
- Klammerfehler
  Im Dokument fanden sich zahlreiche Klammerfehler verschiedener Art: Öffnende Klammern wurden nicht geschlossen, oder es gab zu einer schließenden Klammer keine öffnende. Da es in der Formelsprache verschiedene Klammerarten gibt (runde, eckige, geschweifte, spitze und Schlüsselwörterklammern) wurden etliche Fehler auf syntaktischer Ebene entdeckt, die durch nicht-paarige Verwendung von Klammern (etwa aus dem Vertauschen zweier schließender Klammern) entstanden sind.
- Nicht unterstützte Konstrukte (1)
  Gewisse Konstrukte, die in der formalen Semantik Verwendung fanden, wurden von SDLC nicht unterstützt. Beispielsweise bezeichnet

  [*Identifier*] *

  eine Domäne, die aus einer Folge von optionalen Identifier-Knoten besteht (d.h. jedes Element dieser Folge kann auch *undefined* sein). In diesem und einer Reihe anderer Fälle ist es möglich gewesen, SDLC zu erweitern, so dass das Konstrukt richtig verarbeitet wird.
- Nicht unterstützte Konstrukte (2)
  In anderen Fällen konnte SDLC nicht geeignet erweitert werden. Beispielsweise erlaubt die Formelsprache die Konstruktion

  element **in** *sequence*

  Hierbei ist sequence ein Bezeichner, der eine Folge bezeichnet, und element ist ein potentielles Element dieser Folge. Das ganze Konstrukt ist logisch wahr, wenn das Element in der

Folge vorkommt. Nun hat das Schlüsselwort **in** in der Formelsprache mehrere Bedeutungen. Es wird auch für die folgende Konstruktion verwendet:

```
let variable = expression1 in
    expression2-using-variable
endlet
```

Daraus ergibt sich ein Konflikt für den Bison-basierten Parser: Beim Auffinden des Schlüsselworts **in** ist nicht klar, ob es sich um das zum Let-Ausdruck gehörende Schlüsselwort handelt oder um einen Sequence-Element-Test. Dieser Konflikt wurde zunächst entschärft, indem für die Ausdrücke in einem Sequence-Test-Ausdruck nur einfache ASM-Variablennamen erlaubt wurden, womit die Zahl der möglichen Konfliktfälle reduziert wurde. In den verbleibenden Fällen muss man durch das Setzen zusätzlicher Klammern den Konflikt lösen. Durch diese Einschränkung wurde die Ausdruckskraft der Formelsprache nicht verringert, und die nötigen Korrekturen konnten systematisch erfolgen.

4.2.4 SDLC-Generierungsprobleme

Nachdem SDLC die Formeln verarbeiten kann, generiert er kimwitu++-Dateien. Aus diesen Dateien werden dann mittels Kimwitu++ C++-Quelltext erzeugt. Bei der Generierung dieser C++-Quelltexte stellen Kimwitu++ und der C++-Compiler eine Reihe von Fehlern fest.

Zum einen handelt es sich dabei um Typfehler, wie sie im nächsten Abschnitt erläutert werden. Zum anderen gibt es Konstrukte, die SDLC zwar einlesen (parsen) kann, für die aber kein oder kein korrekter kimwitu++-Code generiert wird. Im Laufe des Projekts traten u.a. folgende Probleme zu Tage:

- Vom SDLC-Generator verursachte C++-Typfehler
  Die beiden abstrakten Syntaxsysteme AS0 und AS1 sind in kimwitu++ durch je eine Grammatikregel (*phylum*), `AS0_rule` und `AS1_rule`, repräsentiert. Müssen nun im generierten Kimwitu++-Code Variablen definiert werden, benötigt der Generator den jeweils richtigen C++-Typ (also `AS0_rule` oder `AS1_rule`) für jeden Ausdruck. Um diese Typisierung in jedem Fall korrekt durchzuführen, müsste SDLC eine Typanalyse vornehmen.
  Von dieser Typanalyse wurde in [Pie00] jedoch Abstand genommen: Erstens ist ein beachtlicher Realisierungsaufwand nötig. Zweitens erhält SDLC gegenwärtig nie alle Typinformationen gleichzeitig, sondern verarbeitet Syntaxdefinitionen, Funktionen, Transformation und Bedingungen separat. Drittens wird dieser Typtest teilweise vom C++-Compiler vorgenommen, so dass gewisse Typfehler in jedem Fall entdeckt werden.
  Statt dessen wurde in [Pie00] die Verwendung einer Heuristik vorgeschlagen. Diese Heuristik ermittelt den C++-Typ einer Variablen durch Inspektion des Namens der Domäne in der formalen Semantik: Fängt der Name mit einem <-Zeichen an (wie etwa <sort>), handelt es sich um einen Wert vom Typ `AS0_rule`. Fängt er mit einem Großbuchstaben an, muss man `AS1_rule` verwenden. Der Generator speichert den jeweils zuletzt beobachteten Typnamen und vermutet, dass alle unmittelbar folgenden Ausdrücke den gleichen Typ haben.
  Wenngleich diese Heuristik für die in [Pri99] beschriebene SDL-Teilsprache ausreichend war, zeigt sie bei Anwendung auf die SDL-2000-Semantik erhebliche Mängel: In manchen Funktionen werden AS0- und AS1-Konstrukte gemischt. In diesem Fall hilft das Raten eines Typs nichts, da die Werte des jeweils anderen Typs dann falsch klassifiziert würden. Zur Lösung dieses Problems wurden verschiedene Strategien realisiert, die die tatsächliche Typisierung dem C++-Compiler überlassen, indem sie polymorphe Typen einsetzen. Diese Strategien werden in Abschnitt 9.4.2 vorgestellt.

- Vom SDLC-Generator verursachte Kimwitu++-Typfehler
  In der abstrakten Syntax (0 und 1) gibt es Regeln, die Vereinigungsmengen darstellen, beispielsweise

  Signal-destination=Expression
  | *Agent-identifier*
  | **THIS**

  Durch diese Regel wird eine Menge *Signal-destination* definiert, die die Vereinigung der Mengen *Expression* und *Agent-identifier* sowie der Einermenge { **THIS** } ist. Diese Vereinigungen werden nun in den Formeln als Muster in **case**-Anweisungen verwendet, beispielsweise in dem Ausdruck

  case *id.parentAS1* **of**
  | *Create-request-node* ∪ *Signal-destination* => **agent**
  | *Agent-type-definition* ∪ *Agent-definition* => **agent type**
  | *Procedure-definition* ∪ *Call-node* ∪ *Value-returning-call-node* => **procedure**
  | *Gate-definition* ∪ *Channel-pathOutput-node* ∪ *Save-signalset* => **signal**
  | *Data-type-definition* ∪ *ParameterResultSignal-definition* ∪ *Timer-definition* ∪
  Exception-definition ∪ *Formal-argumentVariable-definition* ∪ *Any-expression* => **sort**

  ...

  Dieser **case**-Ausdruck wird in eine Kimwitu++-**with**-Anweisung übersetzt, in der die Syntaxregeln als Muster auftauchen und damit Alternativen (Operatoren) der Kimwitu++-Regel AS1_rule sein müssen. Das funktioniert gut für die strukturierten Syntaxregeln, die nicht durch eine Vereinigung, sondern als Tupel definiert sind. Die durch Vereinigung definierten Syntaxregeln werden in Kimwitu++ nicht direkt repräsentiert. Trotzdem hat SDLC diese Regelnamen in einer **with**-Anweisung ausgegeben, etwa als AS1_Signal_destination. Kimwitu++ hat diese Anweisungen dann abgelehnt, da AS1_Signal_destination kein gültiger Operator ist.
  Zur Lösung dieses Problems muss SDLC die Vereinigung in der Generierung für kimwitu++ auflösen, also anstelle von AS1_Signal_destination die drei Alternativen AS1_TOKEN("THIS"), AS1_Agent_Identifier und AS1_Expression generieren (tatsächlich sind auch *Agent-Identifier* und *Expression* wiederum Vereinigungen, die also rekursiv expandiert werden müssen). Zu diesem Zweck wird in SDLC bei der Verarbeitung der Grammatikdefinitionen eine Tabelle generiert, die für jede Vereinigungsregel die Alternativen aufführt. Diese Tabelle wird dann während der Generierung von **case**-Ausdrücken verwendet.
- Nicht unterstützte Konstrukte
  Nicht immer konnten Generierungsprobleme des SDLC-Werkzeugs durch Korrektur dieses Werkzeugs selbst behoben werden. Beispielsweise wurde in Transformationen von Mustern der folgenden Art verwendet

  < <state>(< s >, *exc1*, *triggers1*) > ⌢ *rest* ⌢ < <state>(< *s* >, *exc2*, *triggers2*)>

  Dieses Muster beschreibt eine Folge, die zwei Knoten des Typs <state> enthält, die ihrerseits wieder als ersten Kindknoten einelementige Folgen mit dem gleichen Element s enthalten. exc1, triggers1, exc2 und triggers2 sind jeweils freie Variablen. Zwischen beiden Knoten vom Typ <state> liegen beliebige weitere Elemente, die an die Variable rest gebunden werden.
  In Kimwitu++ werden nun diese Folgen in Listen abgebildet, die durch eine Verkettung von Cons-Zellen[10] entstehen. Das sich ergebende Kimwitu++-Muster müsste nun eine beliebige Zahl von Cons-Zellen überspringen und an die Variable rest binden können; das ist in Kimwitu++ nicht möglich.

Zur Lösung dieses Problems muss eine Reformulierung des Musters gefunden werden. Beispielsweise ist es möglich, durch ein zusätzliches Prädikat zu testen, ob in der Restliste der gleiche Zustand ein weiteres Mal auftaucht:

< <state>(< s >, *exc1*, *triggers1*) > ⌢ *rest*
provided ∃ *s1* ∈ *rest.toSet*: *s1*.**s-**<state list> = < *s* >

Hier wurde das Muster in ein von Kimwitu++ unterstütztes umformuliert und die zusätzliche Bedingung aufgestellt, dass in der Restliste ein Element vorhanden ist, welches in seinem ersten Feld (vom Typ <state list>) den Wert < s > hat (also die Liste, die nur aus dem Element s besteht). Diese Reformulierung reicht in diesem Beispiel noch nicht aus, da die Variablen exc2 und triggers2 nicht mehr gebunden werden. Da diese Variablen auf der rechten Seite der Transformation benötigt werden, muss ihre Berechnung dort separat erfolgen.

4.2.5 Typfehler im Word-Dokument

Sowohl bei der Übersetzung des generierten Kimwitu++-Texts als auch bei der Übersetzung des sich ergebenden C++-Quellcodes zeigten beide Werkzeuge eine große Zahl von Typfehlern an. Sofern diese Fehler sich nicht auf Generierungsfehler zurückführen ließen, mussten sie in den Formeln der Semantikdefinition geändert werden. Dabei traten vor allem die folgenden Fehlerklassen auf:

- C++-Typfehler
  Bei Konstruktion von Knoten der abstrakten Grammatik (0 oder 1) sowie bei Mustertests wurde die falsche Zahl von Argumenten verwendet. Viele dieser Fehler lassen sich darauf zurückführen, dass im Entwicklungsprozess die Grammatik geändert wurde, die Funktionen, die auf den geänderten Regeln basierten jedoch nicht aktualisiert wurden. Beispielsweise wurde zu einem Zeitpunkt in eine Reihe von Grammatikregeln ein weiterer Knoten zur Modellierung von Ausnahmebehandlung eingefügt. Die schon vorhandenen Transformationsregeln wurden dann nicht systematisch aktualisiert. Erst der Typtest im Werkzeug fand diese Probleme.
- Kimwitu++-Typfehler
  Bei der Konstruktion von Knoten wurden Grammatikregeln verwendet, die Vereinigungsdomänen bezeichnen. Dies ist ein Fehler, weil nicht klar ist, welche der möglichen Alternativen denn konstruiert werden sollte. Zur Korrektur muss das jeweilige Problem analysiert werden: Teilweise ist es richtig, eine bestimmte Alternative auszuwählen; teilweise kann dieser Konstruktor-Ruf gänzlich weggelassen werden. Beispielsweise fand sich in den Transformationen das Konstrukt

```
<terminator statement>(undefined,
        <nextstate>(<nextstate body gen name>("WAIT", undefined, undefined))
                )
```

  Der Autor hat bei der Formulierung dieser Konstruktion die Syntaxregel für <terminator statement> studiert:

```
<terminator statement>  :: [<label>] <terminator>
```

  Dabei hat er festgestellt, dass ein Knoten vom Typ <terminator statement> zwei Kindknoten besitzt. Der erste ist optional (kann also undefined sein); der zweite ist vom Typ <terminator>. Verfolgt man die Definition von <terminator>, findet man

10. Der Begriff Cons-Zellen (*cons cells*) entstammt der Sprache Lisp [McC79]. Er bezeichnet ein Objekt mit zwei Attributen: Dem Verweis auf das erste Listenelement und dem Verweis auf die Restliste (ebenfalls wieder eine Cons-Zelle).

<terminator> =<nextstate> | <join> | <stop> | <return> | <raise>

In diesem Fall war die Alternative <nextstate> erforderlich, die wiederum als

<nextstate>=<nextstate body>

definiert ist. Bei flüchtiger Inspektion scheint es, als ob ein <nextstate>-Knoten genau einen Kindknoten enthält, nämlich den Knoten <nextstate body>. Tatsächlich wird durch das Gleichheitszeichen jedoch eine Vereinigungsdomäne ausgedrückt, die in diesem Fall nur eine Alternative enthält. Bei der Konstruktion des Knotens <terminator statement> im Beispiel ist also der Konstruktorruf für <nextstate> wegzulassen.

4.3 Die Abarbeitung der statischen Semantik

Sind die in Abschnitt 4.2 von den Werkzeugen gefundenen Fehler behoben, wird es möglich, eine SDL-Spezifikation an das generierte SDL-Werkzeug zu übergeben und von diesem ein AsmL-Programm generieren zu lassen. Auch hierbei können wiederum verschiedene Probleme aufgedeckt werden.

4.3.1 Fehler in der Definition der abstrakten Syntax 0

Zur Formalisierung der statischen Semantik ist es erforderlich, dass die abstrakte Syntax 0 alle Aspekte der ursprünglichen SDL-Spezifikation erfasst. Falls das nicht der Fall ist, muss die abstrakte Syntax geändert werden.

Beispiel 9. In der konkreten textuellen Syntax treten Schlüsselworte oft paarig auf. Beispielsweise wird eine Prozedurdefinition von **procedure** eingeleitet und von **endprocedure** beendet. Hinter procedure muss der Prozedurname angegeben werden; hinter **endprocedure** kann er angegeben werden. Sind beide Namen angegeben, so müssen sie übereinstimmen. Leider erfasst die abstrakte Syntax 0 nicht den hinter **endprocedure** angegebenen Namen, da er eigentlich redundant ist. Damit kann der Test, ob die Namen gleich sind, nicht implementiert werden. Dieses Problem ist bislang ungelöst; zur Lösung könnte etwa in jeden betroffenen Knoten der abstrakten Syntax 0 ein weiterer Kindknoten eingefügt werden. Diese Lösung hätte allerdings zur Folge, dass jedes Vorkommen der geänderten Knoten angepasst werden müsste.

4.3.2 Überprüfung statischer Bedingungen für die Abstrakte Syntax

Ein Kern der statischen Semantikanalyse ist die Überprüfung von semantischen Regeln (*well-formedness conditions*). Falls diese nicht erfüllt sind, gilt die Eingabespezifikation als falsch. In diesem Fall ist die weitere Verarbeitung der SDL-Spezifikation sinnlos.

Falls angenommen werden kann, etwa durch Inspektion der Spezifikation, dass die Spezifikation allen Regeln entspricht, ist die Überprüfung der Regeln für die Funktion der Werkzeugkette nicht zwingend erforderlich. Nach Analyse der statischen Bedingungen in der formalen Semantikdefinition zeigte sich, dass diese Formeln eine große Zahl der in Abschnitt 4.2 erläuterten Probleme aufweisen, dass also diese Formeln nicht ohne massive Änderungen durch SDLC verarbeitet werden können. Da die Funktion der Werkzeugkette nicht essentiell von diesen Bedingungen abhängt, wurde auf ihre Integration in das Referenzwerkzeug verzichtet.

Die folgende Liste möglicher Probleme ist also spekulativ. Sie ergeben sich lediglich durch Inspektion der Regeln, nicht jedoch durch Überprüfung mit Werkzeugen. Trotzdem erscheint es dem Autor wahrscheinlich, dass bei Integration dieser Bedingungen einige dieser Probleme tatsächlich auftreten würden.

Die potentiellen Fehler in den statischen Bedingungen unterteilen sich in drei Kategorien:

1. Die Regel lehnt richtige Spezifikationen fälschlich ab.

2. Die Regel akzeptiert fälschlich richtige Spezifikationen.
3. Die Regel terminiert für eine gegebene Spezifikation nicht.

Beobachtet man nun, dass eine Spezifiikation abgelehnt wird, so kann das folgende Ursachen haben:

- Fehler in der Spezifikation
  Im Idealfall zeigt das Scheitern eines Prädikats einen tatsächlichen Fehler in der Spezifikation an. Die statische Semantikanalyse leistet dann das Verlangte: Sie zeigt Fehler in der statischen Semantik auf.
- Fehler der Formalisierung
  Ein Prädikat kann auch scheitern, weil die in der informalen Semantik angegebene Regel nicht richtig formalisiert wurde. In diesem Fall muss die Formel korrigiert werden. Eventuell wurde die informale Regel fehlinterpretiert. Dann muss auch die englische Formulierung überarbeitet werden.
- Fehler der informalen Semantik
  Weiterhin kann ein Prädikat scheitern, weil die informale Regel falsch oder zu streng ist. In diesem Fall müssen die Autoren des Sprachstandards entscheiden, ob die formulierte Regel tatsächlich so gemeint war, und ob in Konsequenz also die Eingabespezifikation tatsächlich als falsch gelten soll. Ist das nicht der Fall, muss die Sprachdefinition revidiert werden, sowohl in ihrem informalen als auch in ihrem formalen Teil.
- Fehler im Abarbeitungsmodell
  Schließlich ist es denkbar, dass ein Prädikat nur zu gewissen Zeitpunkten in der Verarbeitungskette erfüllt ist. So sind beispielsweise zahlreiche Prädikate für die abstrakte Syntax 0 formuliert. Die in dieser Syntax vorliegende Eingabe wird jedoch auf verschiedene Weise transformiert, so dass ursprünglich geltende Eigenschaften ungültig werden oder geforderte Eigenschaften erst nach einigen Transformationsschritten gültig werden. Es ist nicht bekannt, ob dieser Fall tatsächlich auftreten kann, oder wie in diesem Fall zu verfahren wäre. Eine mögliche Strategie wäre, für jede dieser Bedingungen anzugeben, nach welchem Transformationsschritt sie zu überprüfen ist. Eine andere Strategie wäre dagegen, zu versuchen, die Bedingungen so umzuformulieren, dass sie, für eine bestimmte Spezifikation, entweder stets oder nie gelten (der genaue Zeitpunkt der Überprüfung also irrelevant ist). Sollte letztere Strategie verfolgt werden, so ist es eine Beweisverpflichtung, nachzuweisen, dass die Bedingungen ihren Wert während der Verarbeitung tatsächlich nicht ändern.

Neben diesen Ursachen für ein Scheitern der Überprüfung von Bedingungen ist es weiterhin möglich, dass Fehler in den Werkzeugen zu einer Fehlevaluierung von Bedingungen führen. Wie in Abschnitt 2.5 erläutert, werden solche Fehler systematischer Art sein und sich beispielsweise auf bestimmte Konstrukte beziehen. Da die in den Bedingungen verwendeten Konstrukte im Wesentlichen die gleichen sind, die auch in den anderen Teilen der formalen Semantikdefinition verwendet werden und da diese schon intensiv getestet wurden, kann man davon ausgehen, nur noch wenige derartiger Probleme zu finden.

Nun kann nicht nur das Scheitern einer Bedingung ein Fehler sein. Die Berechnung der Bedingung könnte auch fälschlich den Wert „wahr“ liefern, oder sie könnte nicht terminieren. Der erste Fall einer fälschlich erfüllten Bedingung lässt sich nur schwer entdecken. Gibt man eine Spezifikation ein, von der nicht bekannt ist, ob sie richtig oder falsch ist, kann man nur erhoffen, dass eine falsche Spezifikation im Laufe des Verarbeitungsprozesses noch weitere Fehler (sogenannte Folgefehler) provoziert. Die Analyse der Folgefehler kann dann zu der Einsicht führen, dass diese Spezifikation gar nicht hätte verarbeitet werden dürfen. In manchen Fällen wird diese Information vorab bekannt sein, beispielsweise wenn ein anderes Werkzeug die Spezifikation bereits abgelehnt hat. Dann ist zu untersuchen, ob das Problem in der Formalisierung der Bedingung, in der Bedingung selbst oder aber in dem anderen Werkzeug liegt.

Falls die Berechnung einer Bedingung nicht terminiert, so wird das hingegen relativ schnell offensichtlich: die Verarbeitung der Spezifikation dauert bedeutend länger als die Verarbeitung „vergleichbarer“ Spezifikationen. In diesem Fall ist zunächst zu untersuchen, ob nicht vielleicht doch noch mit einer Terminierung der Berechnung zu rechnen wäre und lediglich ein Algorithmus zum Einsatz kommt, der eine große Komplexität besitzt. Auch wenn dieses Problem als das Halteproblem bekannt ist [Tur36], und damit im Allgemeinen unlösbar ist, hat der Autor doch die Erfahrung gemacht, dass sich in der Praxis diese Frage immer leicht beantworten lässt. Ursache hierfür ist die kompositionale Struktur der Übersetzung der Formeln: Jede Formel besteht aus Ausdrücken, für die oft klar ist, ob sie stets terminieren oder nicht. Die Berechnung der Gesamtformel terminiert bestimmt dann, wenn alle Teilausdrücke terminieren und der Kompositionsoperator auch stets terminiert. Eine Ausnahme sind natürlich rekursiv definierte Funktionen, für die oft nicht so einfach zu beantworten ist, ob sie terminieren – diese sind aber relativ selten, und oft auch in ihrer Rekursion überschaubar.

Unter den in der formalen Semantik verwendeten Konstrukten gibt es einige, die potentiell nicht terminieren. Insbesondere terminiert die Berechnung von Quantoren unter Umständen nicht, wenn die Basismenge unendlich ist. Gerade in den Bedingungen wird stark Gebrauch von Quantoren gemacht. Beispielsweise wird die Forderung

Exactly one of the <start>s shall be unlabelled.

durch die Formel

$\forall$ *csg* $\in$ <composite state body>:

$\exists!$*s* $\in$ <start>: (*s.parentAS0* = *csg*) $\wedge$ (*s*.**s**-<name> = *undefined*)

formalisiert: Für alle Knoten csg des Typs <composite state body> (auf die sich Forderung bezog) gibt es je genau einen Knoten <start>, dessen Elternknoten csg ist und dessen <name>-Kind undefiniert ist. Zur Überprüfung dieser Regel müssen alle Knoten des Typs <start> inspiziert werden. Es reicht nicht, abzubrechen, wenn ein derartiger Knoten gefunden wurde. Diese Berechnung terminiert, weil die Menge aller <start>-Knoten in einer Spezifikation endlich ist.

Es gab in der Semantikdefinition allerdings auch Fälle, in denen eine Quantifizierung unendlicher Mengen vorgenommen wird, beispielsweise Forderungen der Art „Es gibt eine Zahl, so dass ...“ Wenn diese Forderungen durch Quantifizierung über die Menge der natürlichen Zahlen formalisiert wurden, würde die Berechnung nicht terminieren. Der Generator SDLC erkennt eine Reihe solcher Fälle und zeigt sie als Fehler an.

4.3.3 Die Abarbeitung von Transformationsregeln

Die Transformationsregeln, die einen Baum der abstrakten Syntax in einen anderen umformen, können keine Fehler im eigentlichen Sinn produzieren. Trotzdem wurden beim Test der Werkzeuge folgende Fehlerfälle erkannt:

- Endlosschleifen
  Eine Transformation wird immer wieder ausgeführt, weil der Baum, der am Ende der Transformation entsteht, wieder die Vorbedingung der Anwendung der Transformationsregel erfüllt. Dabei ist sowohl der Fall beobachtet worden, dass der Baum im Rahmen der Transformation nicht verändert wurde, als auch, dass er vergrößert wurde. Im ersten Fall liegt der Fehler klar in der Formalisierung der Transformationsregel, da die formalisierte Version offenbar überhaupt keinen Effekt hat. Im zweiten Fall muss vielleicht auch die englische Formulierung überarbeitet werden, falls sie für den Fall des rekursiven Anwendens der Regel keine klaren Regelungen trifft.
- Auslassen von Transformationen
  Eine Transformation, die ausgeführt werden müsste, wird nicht ausgeführt. Dies fällt in der

Regel erst in späteren Verarbeitungsschritten auf, wenn etwa die Abbildung auf die Abstrakte Syntax 1 Knoten in der Baumstruktur vorfindet, für die keine Abbildung vorgesehen ist, weil sie eigentlich durch eine Transformation hätten ersetzt werden müssen. Die Ursachen für dieses Problem können vielfältig sein, so kann es beispielsweise erforderlich sein, die Reihenfolge der Tranformationen zu ändern oder die Vorbedingungen für die Regel zu korrigieren.

4.3.4 Die Generierung und Verarbeitung von AsmL

Nach Abarbeitung der Transformationen und Konstruktion der abstrakten Syntax 1 wird durch die Kompilationsfunktion AsmL-Quelltext erzeugt. Dieser Quelltext wird dann vom AsmL-Compiler weiterverarbeitet und schließlich durch das Microsoft .NET-Laufzeitsystem ausgeführt.

Bei der Übersetzung des AsmL-Quelltextes waren die Probleme vorwiegend folgender Natur:

- Von AsmL nicht unterstützte Konstrukte
  Die erste Version des AsmL-Compilers hat eine Reihe von Konstrukten nicht unterstützt, die in der SDL-Semantikdefinition an zahlreichen Stellen verwendet wurden (beispielsweise Vereinigungsdomänen). Daraufhin wurde von der Verwendung von AsmL 1.5 Abstand genommen und statt dessen mit einer Beta-Version von AsmL 2 gearbeitet, die im Wesentlichen alle benötigten Konstrukte bereitstellt. Manche Aspekte der SDL-Semantik werden von AsmL gar nicht unterstützt, etwa die Verteilung des Systems auf mehrere Agenten oder der Begriff der Systemzeit. Um diese Aspekte auf AsmL abzubilden, wurde eine zentrale Ablaufsteuerung geschaffen, die Zeit- und Agentensemantik nachbildet.
- Typfehler im AsmL-Code
  Der AsmL-Compiler hat Typfehler gemeldet. Hauptursache des Problems ist, dass AsmL getypte Variablen verwendet, während die SDL-Semantikdefinition ungetypte Variablen und Funktionen verwendet (alle Funktionen sind partiell auf einer Grundmenge X definiert und liefern den Wert *undefined* für Argumente außerhalb ihres eigentlichen Definitionsbereichs). Glücklicherweise wurde von diesem Verzicht auf Typisierung nur an wenigen Stellen Gebrauch gemacht, so dass durch Einführung von Typdeklarationen an wenigen Stellen das Typsystem von AsmL zufrieden gestellt werden konnte.
- Fehler im AsmL-Compiler
  Der AsmL-Compiler hat Fehler gezeigt (beispielsweise Abstürze). Diese Probleme mussten zusammen mit den Autoren des Compilers behoben oder umgangen werden.

4.4 Abarbeitung des generierten AsmL-Programms

Auch bei der Abarbeitung des aus dem SDL-System entstandenen AsmL-Programms traten in den Beispielen viele Fehler auf, die sich in der Regel durch Programmabbruch dargestellt haben. In diesen Fällen ist die mögliche Fehlerursache jeder der vorangegangen Verarbeitungsschritte. In der Praxis zeigte sich, dass Programmabbrüche oft durch Fehler in der Definition der dynamischen Semantik verursacht wurden.

4.5 Fehler der informalen Semantikdefinition

Während ein großer Teil der Fehler sich durch die Korrektur der formalen Semantikdefinition beheben ließ, sind bei der Definition der formalen Semantik auch einige Probleme im Haupttext des SDL-Standards entdeckt worden. Diese lassen sich wiederum in verschiedene Gruppen zusammenfassen, für die hier Beispiele angegeben werden sollen.

4.5.1 Mehrdeutigkeit der konkreten Syntax

Da gleichzeitig mit der Entwicklung von SDL-2000 auch das Werkzeug SDLC entwickelt wurde, ergab sich die Notwendigkeit, einen Parser für die konkrete Syntax zu konstruieren. Dieser Parser entstand auf Basis von Bison [FSF02a]. Die aus dem Entwurf von SDL-2000 extrahierte Grammatikdefinition hat für Bison Hunderte von Konflikten erzeugt. Diese Konflikte hatten verschiedene Ursachen:

- Die Grammatikformulierung im Sprachstandard dient der Erläuterung der Sprache, nicht der Konstruktion von Parsern. Es handelt sich um eine Präsentationsgrammatik, die bewusst Redundanzen enthält, die die Lesbarkeit erhöhen sollen. Dies führt dazu, dass die Grammatik einfach zu beseitigende Mehrdeutigkeiten enthält.
- Der von Bison verwendete LALR(1)-Algorithmus meldet Konflikte, obwohl die Grammatik nicht mehrdeutig ist. Manche dieser Konflikte kann man ebenfalls durch Umformulierung beheben, andere nur durch Erweiterung der akzeptierten Sprache und Einschränkung auf die Originalsprache durch semantische Aktionen.
- Die (um triviale Mehrdeutigkeiten bereinigte) SDL-Grammatik ist tatsächlich auf eine Weise mehrdeutig, die nicht durch eine Umformulierung beseitigt werden kann. Beispielsweise ist in dem Konstrukt „output sig to p“ nicht klar, ob „p“ ein Ausdruck oder der Name eines Prozesses ist. Diese Konflikte werden in SDL durch zusätzliche englische Erklärungen behoben, die bei Mehrdeutigkeiten einer der Interpretationen Vorzug geben. Eine Beseitigung dieser Mehrdeutigkeiten ist wegen Beachtung der Rückwärtskompatibilität nicht möglich.

Zusätzlich zu diesen Schwierigkeiten zeigte sich, dass auch durch die Erweiterung von SDL-92 zu SDL-2000 neue Konflikte in die Sprache eingeführt wurden. In der Regel erfolgte dies unabsichtlich. Durch den Einsatz von Bison konnten einige dieser Konflikte aufgedeckt werden. Das Normierungsgremium musste dann entscheiden, ob der Konflikt durch Korrektur der Grammatik oder durch zusätzliche Erklärungen beseitigt wird. Als Beispiel soll hier der Konflikt zwischen „verkürzten Datentypdefinitionen“ und „verschachtelten Datentypdefinitionen vorgestellt werden.

Für SDL-2000 gab es einerseits den Wunsch, Datentypdefinitionen innerhalb von Datentypdefinitionen zu erlauben, wie folgendes Beispiel demonstriert.

```
object type Aussen;
   object type Innen;
      literals rot, gruen, blau;
   endobject type;
   struct
      feld1 Integer;
      feld2 Innen;
endobject type;
```

Gleichzeitig gab es den Wunsch, den syntaktischen Aufwand für eine eigentlich einfache Typdefinition zu reduzieren, also auf das schließende Konstrukt **endobject type** verzichten zu können. Dazu wurde der Vorschlag unterbreitet, es einfach wegzulassen, und beispielsweise schreiben zu dürfen

```
object type Punkt;
   struct x,y,z Real;
```

In diesem Konstrukt ist klar, dass die Strukturdefinition zu dem Typ Punkt gehört. Durch die Angabe der dann folgenden Schlüsselwörtern müsste dann auch klar werden, dass die Typdefinition abgeschlossen ist. Dieser Vorschlag erzeugte allerdings eine Mehrdeutigkeit. Nimmt man die Definitionen

```
object type Aussen;
object type Innen;
   struct
      feld2 Innen;
```

so ist zwar klar, dass einer der Typen eine leere Definition hat, aber nicht, welcher: Der Typ Aussen könnte als leer interpretiert werden, in welchem Fall Innen ein Strukturtyp wäre, oder es könnte der Typ Aussen als Strukturtyp verstanden werden, dann wäre Innen ein leerer Typ (tatsächlich ist noch eine dritte Interpretation möglich: Innen könnte als nicht-verschachtelte Definition betrachtet werden).

Dieser Konflikt wurde beseitigt, indem zur Kurznotation von Typen eine gänzlich andere Syntax eingeführt wurde: In SDL-2000 können nun geschweifte Klammern zur Gruppierung von Definitionen verwendet werden, so dass der Typ Punkt als

```
object type Punkt{
   struct x,y,z Real;
}
```

notiert werden kann.

4.5.2 Unvollständige Definition der abstrakten Syntax

Im Haupttext des SDL-Standards wird neben der konkreten Syntax auch die abstrakte Syntax (Abstrakte Syntax 1) definiert. Der informale englische Text nimmt Bezug auf diese Definition, indem die Semantik auf Basis der abstrakten Syntax erläutert wird.

In der formalen Semantik bilden die Regeln der abstrakten Syntax den Abschluss der statischen Semantikdefinition. Nur die Informationen, die in der abstrakten Syntax festgehalten sind, stehen in der Definition der dynamischen Semantik zur Verfügung.

Bei der Definition der formalen Semantik ist nun aufgefallen, dass manche Informationen, die in der dynamischen Semantik benötigt werden, nicht in der abstrakten Syntax definiert sind, und damit die dynamische Semantik nicht korrekt formalisiert werden kann.

Beispielsweise sieht ein Transformationsmodell für Operatordefinitionen vor, dass für jeden Operator eine anonyme Prozedur definiert und danach die Operatordefinition verworfen wird. Operatorrufe sind so semantisch Prozedurrufen gleichwertig.

Weiterhin sieht das objektorientierte Datentypmodell vor, dass bei einem virtuellen Methodenruf die gerufene Prozedur spät gebunden wird. Das heißt, dass in dem Knoten der abstrakten Syntax, der einen Methodenruf repräsentiert, der Name der Prozedur noch nicht festgehalten werden kann, sondern nur der Name der Operationssignatur, die dynamisch gebunden werden soll. Diese war zunächst durch

```
Operation-signature::   Operation-name
                        Formal-argument*
                        [Result]
```

definiert. Da sich auch hier kein Verweis auf die zu rufende Prozedur fand, wurde auf Vorschlag des Autors diese Regel neu formuliert als

```
Operation-signature::   Operation-name
                        Formal-argument*
                        [Result]
                        Identifier
```

Zahlreiche solcher Änderungen konnten in die 1999 verabschiedete Fassung von SDL-2000 oder deren Aktualisierung [Z.100-01] integriert werden. Bei einigen Knoten der abstrakten Syntax steht diese Integration jedoch noch aus, so ist beispielsweise zur Realisierung der dynami-

schen Semantik von Schnittstellen eine Repräsentation von Schnittstellen in der abstrakten Syntax erforderlich. Diese ist derzeit nicht im Haupttext des SDL-Standards zu finden.

4.5.3 Mehrdeutigkeiten der Semantik

Zur Definition der formalen Semantik wird die informale Semantik als Ausgangspunkt verwendet, und die dort formulierte Bedeutung eines Konstrukts in das formale Kalkül übertragen. Die formalisierte Fassung wird stets unzweideutig sein (zumindest bei dem für SDL-2000 verwendeten Kalkül der Abstract State Machines), da der Kalkül für einen konkreten Systemzustand des SDL-Systems stets festlegt, was die Menge der möglichen Folgezustände ist. Es ist dabei zwar möglich, dass auf einen Zustand mehrere alternative Zustände folgen können, aber auch in diesem Fall definiert der Kalkül klar die Semantik: von den Folgezuständen wird ein bestimmter nichtdeterministisch ausgewählt.

Für die informale Semantik kann man á priori keine Garantie der Eindeutigkeit abgeben. Es ist durchaus möglich, dass die informale Semantik für gewisse Systemzustände keine Festlegung trifft, welcher von mehreren Abläufen normkonform ist. In diesem Fall ist die informale Semantik mehrdeutig und somit fehlerhaft.

Diese Mehrdeutigkeiten fallen potenziell im Prozess der Formalisierung auf, da der Entwickler der formalen Semantik einen Algorithmus entwerfen muss, der die informalen Festlegung umsetzt. Dabei wird eventuell festgestellt, dass es für diesen Algorithmus mehrere sinnvolle (also in sich widerspruchsfreie) Alternativen gibt. Es ist allerdings auch möglich, dass dem Entwickler dieses Problem nicht auffällt, und er aus den möglichen Alternativen willkürlich und unbewusst diejenige auswählt, die ihm am sinnvollsten erscheint. Im Ergebnis weicht dann die formale Semantik von der informalen ab. Für SDL wurde festgelegt, dass in diesem Fall das Normierungsgremium diesen Konflikt bereinigen muss (also keine der beiden Formulierungen der Semantik automatisch Vorrang bekommt).

In einer Reihe von Fällen konnten die Mehrdeutigkeiten der informalen Semantik jedoch vor Verabschiedung von SDL-2000 aufgelöst werden. Ein Beispiel hierfür ist die Semantik des **decision**-Konstrukts für den Fall, dass der Vergleich von Werten eine Ausnahme liefert. Mit diesem Konstrukt kann man, abhängig von der Berechnung eines Ausdrucks, einen von mehreren Kontrollflüssen eines Agenten auswählen. Syntaktisch stellt sich dieses Konstrukt wie folgt dar:

```
decision Ausdruck;
(Antwort1): Aktion1;
(Antwort2): Aktion2;
else: Aktion3;
enddecision;
```

In diesem Konstrukt wird zunächst der Ausdruck evaluiert und danach diejenige Aktion ausgeführt, für die die Antwort gleich dem Ausdruck ist[11]. Falls keine der Antworten zutrifft, wird die **else**-Aktion ausgeführt. In SDL-92 war dazu die Termäquivalenz des Vergleichs vom Ausdruck mit der Antwort zum Term True zu überprüfen. Es wurde statisch gefordert, dass höchstens einer dieser Vergleiche wahr wird.

In SDL-2000 zeigt sich, dass die von SDL-92 übernommene Formulierung der Semantik unzureichend ist. Die Bewertung von Ausdrücken und der Vergleich ist nun nicht mehr durch Termäquivalenz definiert, sondern durch Abarbeitung eines Algorithmus – Ausdrücke werden „berechnet“. Diese Berechnung muss nun aber nicht mehr zwingend als Ergebnis True oder False liefern, so könnte auch eine Ausnahme auslösen oder gar nicht terminieren. Die Frage ist,

11. Tatsächlich kann jede Antwort eine Menge möglicher Werte umfassen. Es reicht dann, wenn ein Wert übereinstimmt.

welche der Aktionen dann ausgeführt wird. Mögliche Alternativen des Verhaltens sind die folgenden:

- Der Ausdruck wird mit allen Antworten verglichen, und zwar mit allen gleichzeitig. Liefert eine Berechnung eine Ausnahme, so wird diese verworfen. Sowie eine Berechnung den Wert true liefert, werden die anderen Berechnungen abgebrochen.
- Die Berechnungen werden in willkürlicher Reihenfolge durchgeführt, und zwar alle. Ausnahmen werden verworfen. Falls eine Berechnung nicht terminiert, terminiert auch die Abarbeitung des Decision-Konstrukts nicht. Sollten mehrere Alternativen den Wert True liefern, wird ein Laufzeitfehler ausgelöst.
- Die Berechnungen werden in beliebiger Reihenfolge durchgeführt, bis eine Ausnahme auftritt oder eine Berechnung den Wert true liefert. Im Fall einer Ausnahme wird keine der Aktionen ausgeführt, sondern die Ausnahmebehandlung initiiert.

Nach Diskussion dieser und anderer Alternativen im Normierungsgremium wurde die letzte dieser Alternativen ausgewählt und sowohl die informale Semantik als auch die formale Semantik entsprechend angepasst.

4.5.4 Widersprüche in der Semantik

Neben Mehrdeutigkeiten, die also mehrere alternative Interpretationen des Standards erlauben, können in einer informalen Semantikdefinition auch Widersprüche oder andere Inkonsistenzen enthalten sein, die beispielsweise dazu führen, dass eine gewisse Spezifikation gar nicht abgearbeitet werden kann, weil das System mehrere Bedingung einhalten muss, die gleichzeitig gar nicht erfüllt sein können.

Eine andere Konstellation eines Widerspruchs sind irrelevante Festlegungen: Die Semantikdefinition legt ein Verhalten für einen Fall fest, der gar nicht auftreten kann. Ein Beispiel dafür ist die Definition der Zugriffsmethoden für Strukturtypen. Gegeben sei der Typ

```
object type Base{
    struct
        x Integer;
        y this Base;
}

object type Derived inherits Base{
}
```

Hier werden zwei Strukturtypen definiert, wobei die zweite eine Spezialisierung der ersten ist. Beide besitzen je zwei Felder. Während das Feld x in beiden Strukturen vom Typ Integer ist, unterscheidet sich der Typ des Felds y durch Verwendung des Schlüsselworts **this**: Im Typ Base ist es selbst auch vom Typ Base; im Typ Derived ist es jedoch vom Typ Derived. Die Semantikdefinition legt nun fest, dass es im Typ Base eine Zugriffsmethode

```
virtual yModify(this Base) -> Base;
```

und im Typ Derived eine Methode

```
virtual yModify(this Derived) -> Derived
```

gibt. Die Semantik dieser Methoden wird durch den Text [Z.100-00, 12.1.7.2]

> *The implied method to modify a field associates the field with the result of its argument Expression. When <field sort> was an <anchored sort>, this association takes place only if the dynamic sort of the argument Expression is sort compatible with the <field sort> of this field. Otherwise, the predefined exception UndefinedField (see 3.16) is raised.*

definiert. In dem Beispiel ist der Knoten <field sort> tatsächlich von der Form <anchored sort> (siehe Abschnitt 6.5). Dies wird durch das Schlüsselwort **this** angezeigt. Nun kann es nach dieser Semantikdefinition passieren, dass die Ausnahme UndefinedField auftritt, falls nämlich das Argument der Methode yModify nicht typkompatibel mit dem Parametertyp ist.

Dieser Fall kann allerdings überhaupt nicht auftreten: Damit der dynamische Typ des Arguments nicht kompatibel mit der Sorte des Felds ist, muss der Aufruf der Form

```
dcl d Derived, b Base;
task d := (. 10, null .);
task b := (. 3, null .);
task d := d.yModify(b);
```

sein, das heißt, sowohl der statische als auch der dynamische Typ des Arguments müssen Base sein. In diesem Fall wird jedoch nicht die Methode der Spezialisierung, sondern die Methode des Basistyps aufgerufen, für die das Argument durchaus typkompatibel mit dem Feld ist.

Dieses Problem ist in der verabschiedeten Version von SDL-2000 nicht gelöst worden. Zur Lösung müsste zunächst geklärt werden, was die beabsichtige Semantik dieses Beispiels ist.

4.6 Fazit

Die Entwicklung der formalen SDL-Semantik wäre ohne die Unterstützung durch Werkzeuge nicht möglich gewesen. Die informale Sprachdefinition, die formale Sprachdefinition und die Werkzeuge sind zusammen entwickelt worden, um die Auswirkungen einer Änderung in einem dieser Teile frühzeitig studieren zu können. Durch den Einsatz der Werkzeuge sind vor allem Fehler in der formalen Sprachdefinition entdeckt worden, aber auch Widersprüche und Unvollständigkeiten in der informalen Sprachdefinition.

5 Übersicht über die formale Definition von SDL

Bei der Definition der formalen Semantik wurden folgende Entwurfsziele verfolgt [EGG+01]:
- Verständlichkeit,
- Prägnanz,
- Korrektheit (Übereinstimmung mit der informalen Semantikdefinition) und
- Wartbarkeit.

Um diese Ziele zu erreichen, ist es zum einen nötig, eine klare Relation zwischen der informalen (englischen) Sprachdefinition und der formalen Definition aufrechtzuerhalten. Zum anderen ist es nötig, die Ausführbarkeit der formalen Definition zu ermöglichen, d.h., bei der Auswahl der Formalismen und deren Einsatz stets darauf zu achten, dass aus der formalen Definition auch ein ausführbares Programm abgeleitet werden kann.

Im folgenden wird dargestellt, wie die formale Definition strukturiert ist, und wie dabei die Relation zur informalen Definition und die Ausführbarkeit erreicht wurden. Die Auswahl der Formalismen und ihr Zusammenspiel wird wesentlich in [Pri99] begründet; dieses Kapitel stellt lediglich eine Zusammenfassung der Ideen aus [Pri99] und ihre Weiterentwicklung im veröffentlichten SDL-Standard dar.

5.1 Struktur der informalen Definition

Der Hauptteil der ITU-T-Empfehlung Z.100 [Z.100-00] besteht neben Einleitungen (die auch die verwendeten Notationen erklären) aus folgenden Abschnitten (in Klammern einige Unterabschnitte):
- allgemeine Regeln (lexikalische Regeln, Darstellungsregeln für Diagramme und Definition des Makrokonzepts),
- Organisation von SDL-Spezifikationen (*Package*, *referenced definition*)
- strukturelle Konzepte (Typ, Instanz, Kontextparameter, Spezialisierung),
- Agenten (System, Block, Prozess),
- Kommunikation (Kanal, Signal, entfernte Prozedur),
- Verhalten (Start, Zustand, Transition, Aktion, Timer, Ausnahme) und
- Daten (Datentypen, Schnittstellen, Operationen, Pid-Typen, Literale, Strukturen, Choices, Verhalten von Operationen, Ausdrücke, Variablen).

Für jedes Sprachelement wird die informale Semantik auf einheitliche Weise definiert. Diese Definition besteht aus:
- dem Verwendungszweck des Elements,
- der abstrakten Grammatik,
- Bedingungen für die abstrakte Grammatik,
- der konkreten textuelle Grammatik,
- den Bedingungen für die konkrete textuelle Grammatik,
- der konkreten grafische Grammatik,
- den Bedingungen für die konkrete grafische Grammatik,
- der Semantik sowie
- dem Transformationsmodell.

Jeder dieser Teile ist optional.

5.2 Struktur der formalen Definition

Die formale Definition von SDL [Z.100.F] gliedert sich in drei Teile:

- Teil 1 definiert die verwendeten Notationen, erklärt ihre Bedeutung, und gibt eine Einführung in den Kalkül von Abstract State Machines.
- Teil 2 definiert die statische Semantik, d.h. einen Algorithmus zum Aufbau eines Baums der abstrakten Syntax, sowie eine Menge von Prädikaten, die die Wohlgeformtheitsregeln definieren.
- Teil 3 definiert die dynamische Semantik, d.h. das Verhalten eines Systems (in Form von Zustandsänderungen) bei der Interpretation.

5.3 Teil 1: Übersicht, verwendete Notationen und Kalküle

In Teil 1 der formalen Semantikdefinition wird eine Übersicht über die formale Semantik gegeben. Dazu werden die verwendeten Kalküle eingeführt, insbesondere das der *Abstract State Machines* (ASMs) und die vordefinierten Namen sowie die Notationen, mit der Formeln für diese Kalküle formuliert werden. Die nun folgende Darstellung von ASMs gliedert sich in drei Teile:

1. Abstract State Machines: Zur Definition von ASMs werden die Begriffe Vokabular, Zustand, Interpretation, Aktion und Programm eingeführt. Dieser Begriff „elementarer“ ASMs wird auf verteilte ASMs erweitert und dazu die Begriffe ASM-Agent und Systemzeit eingeführt.
2. Vordefinierte Namen: In dem Vokabular, dass zur Definition der SDL-Semantik verwendet wird, sind einige Namen vordefiniert, die im zweiten Abschnitt erläutert werden.
3. Grammatiken: Zur Repräsentation von abstrakten Syntaxbäumen wird eine Beziehung zwischen Grammatiken und ASMs benötigt. Zu diesem Zweck werden die ASM-Notationen um Konstrukte erweitert, die üblicherweise zur Definition von Grammatiken verwendet werden, und eine Abbildung von diesen Konstrukten auf das ASM-Kalkül vorgestellt.

5.3.1 Abstract State Machines

Der Kalkül der abstrakten Zustandsmaschine (*Abstract State Machine*) [Gur95] ist von Юрий Гуревич (*Yuri Gurevich*) entwickelt worden, ursprünglich unter dem Namen Evolving Algebras [Gur97]. Er bildet die formale Fundierung der SDL-Semantik.

Eine abstrakte Zustandsmaschine M ist für ein *Vokabular V* durch ihre *Zustände* S, ihre *Initialzustände* $S_0 \subseteq S$ und ihr *Programm P* definiert. Das Vokabular enthält die vordefinierten Namen sowie die Namen, die für die Definition von M eingeführt wurden. Diese Namen werden durch Deklarationen eingeführt. Deklariert werden können Domänen und Funktionen (darin sind Prädikate eingeschlossen). Zur Deklaration einer Domäne kann ein Modus angegeben werden, zur Deklaration einer Funktion ein Modus sowie die Argument- und Ergebnisdomänen. Mögliche Modi im ASM-Kalkül sind **static**, **controlled**, **shared** und **monitored**. Ist kein Modus angegeben, so gilt der Name als abgeleitet (*derived*). Die Angabe des Modus bezieht sich auf die Abhängigkeit der Interpretation eines Namens vom Zustand der Maschine.

In jedem Zustand hat jeder Name eine *Interpretation* (also eine Bedeutung). Diese Bedeutung bezieht sich auf die Grundmenge der Maschine M, im folgenden *X* genannt. Eine Interpretation legt für jedes Element x von *X* und jede Domäne *D* fest, ob *x* in *D* enthalten ist. Die Interpretation legt für jede Funktion f und jede Parameterkombination von Werten aus X fest, welchen Wert f für diese Parameterkombination hat.

Diese Interpretation liefert für jeden Funktionsnamen eine totale Funktion: Partielle Funktionen können dadurch repräsentiert werden, dass sie den vordefinierten Wert undefined liefern.

Die Interpretation eines Namens kann sich von Zustand zu Zustand ändern, sofern der Name nicht den Modus **static** hat. Wie diese Änderung erfolgt, hängt vom Modus ab:

- Interpretationen von Namen mit dem Modus **controlled** können durch das Programm P der Maschine geändert werden. Sie repräsentieren den abstrakten Zustand der Maschine.
- Interpretationen von Namen mit dem Modus **monitored** können sich „spontan“ von Zustand zu Zustand ändern. Damit kann die Zustandsmaschine ihre Umgebung beobachten. Das Verhalten der Umgebung kann durch Integritätsbedingungen eingeschränkt werden.
- Interpretation von Namen mit dem Modus **shared** können sowohl von der Umgebung als auch von der Maschine geändert werden. Dabei dürfen die Änderungen nicht im Widerspruch stehen. Die Umgebung darf also einer Funktion für ein Argument keinen anderen Wert zuweisen als die Zustandsmaschine das auch tun würde.

Abgeleitete Namen werden durch Berechnungsformeln definiert. Ihre Interpretation hängt von der Interpretation der Namen ab, auf denen ihre Definition beruht.

Die Änderungen des Zustands werden (sofern sie nicht durch die Umgebung erfolgen) durch das ASM-Programm festgelegt. In jedem Zustand berechnet die Maschine eine endliche Menge von Änderungen der Interpretation von Namen, die sogenannte Update-Menge (*update set*). Genauer gesagt wird die Interpretation eines Namens an einer *Stelle* (*location*) durchgeführt, die durch die Parameterliste bestimmt ist. Die Änderungen werden in einem Schritt ausgeführt. Dadurch entsteht der nächste Zustand der Maschine. Die Änderungen werden in Form von Anweisungen im ASM-Programm notiert. Dazu stehen folgende Anweisungen zur Verfügung:

- Die Update-Anweisung $f(t_1, t_2, ..., t_n) := t_0$

  Mit dieser Anweisung wird die Interpretation von f an der Stelle $t_1, t_2, ..., t_n$ auf den Wert t_0 geändert.
- Die bedingte Anweisung

  if g **then**
  Anweisung$_1$
  [**else**
  Anweisung$_2$]
  endif

  Mit dieser Anweisung wird Anweisung$_1$ ausführt, wenn die Bedingung g gilt, ansonsten die optionale Anweisung$_2$. Fehlt diese wenn g nicht erfüllt ist, so hat die Anweisung keinen Effekt.
- Die Parallel-Anweisung

  do-in-parallel
  Anweisung$_1$
  ...
  Anweisung$_n$
  enddo

  Mit dieser Anweisung werden eine Menge von Anweisungen gleichzeitig ausgeführt.
- Die Forall-Anweisung

  do forall v: g(v)
  Anweisung$_1$(v)
  enddo

  Diese Anweisung wählt alle Elemente v von *X* aus, für die das Prädikat g(v) erfüllt ist, und führt für diese Elemente gleichzeitig die Anweisung$_1$ aus; die Bedingung darf nur für endlich viele Elemente erfüllt sein.
- Die Choose-Anweisung

  choose v: g(v)
  Anweisung$_1$(v)
  enddo

Durch diese Anweisung wird willkürlich ein Element v ausgewählt, für das g(v) erfüllt ist, und die Anweisung$_1$ für dieses Element abgearbeitet. Gibt es kein solches Element, so hat die Anweisung keinen Effekt.

- Die Extend-Anweisung

  extend *D* **with** v_1, ..., v_n
  Anweisung(v_1, ... v_n)
  endextend

  Die Domäne D wird dynamisch um neue Elemente aus *X* erweitert, für die dann die Anweisung$_1$ abgearbeitet wird.

Zusätzlich zu diesen Basiskonstrukten werden im Teil F.1 eine Reihe von Abkürzungen für häufig auftretende Anweisungen definiert, die für die weiteren Ausführungen nicht von Bedeutung sind.

Durch Ausführung von parallelen Anweisungen (**do-in-parallel, do forall**) kann es passieren, dass mehrere Update-Anweisungen zueinander im Konflikt stehen, also einer Funktion an einer Stelle verschiedene Werte zuweisen. Falls das passiert, wird die gesamte Update-Menge ignoriert, das heißt der Nachfolgezustand ist gleich dem Vorgängerzustand. Dieser Fall wird als Fehler im ASM-Programm betrachtet; das Programm wird jedoch trotzdem weiter ausgeführt (wobei im nächsten Schritt unter Umständen wieder der gleiche Konflikt auftritt, so dass sich der Zustand der Maschine nicht mehr ändert). Ein Ablauf eines ASM-Programms beginnt mit einem Initialzustand und setzt sich durch wiederholte Ausführung des ASM-Programms fort: in jedem Interpretationsschritt wird die Update-Menge bestimmt und die Interpretation der Funktionen geändert.

Zur Definition eines verteilten Systems wird der bisher vorgestellte Begriff der abstrakten Zustandsmaschine auf den einer verteilten Echtzeit-ASM ausgeweitet (*distributed real-time ASM*). Eine solche Maschine besteht aus endlich vielen *ASM-Agenten*, von denen jeder ein ASM-Programm ausführt. Die Domäne *AGENT* bezeichnet die Menge aller Agenten. Die Erweiterung von *AGENT* durch eine Extend-Anweisung führt zur Einführung neuer ASM-Agenten. Eine Funktion Self (mit dem Modus **monitored**) bezeichnet innerhalb des ASM-Programms den Agenten, der das Programm ausführt. Abbildung 8 stellt den Zusammenhang zwischen

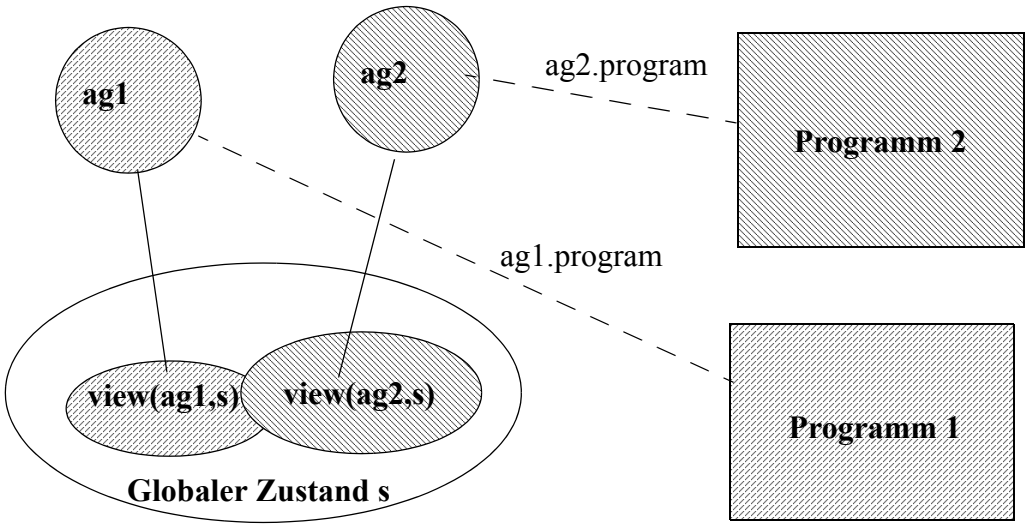

Abbildung 8: Verteilte Echtzeit-ASMs

Agenten, ihren Programmen, und dem Systemzustand dar. Jedem Agenten ist über die Funktion program sein Programm zugeordnet. In diesem Programm erfolgt durch verschiedene Funktionen der Zugriff auf den Systemzustand. In der Regel wird jeder Agent nur einen Teil des Systemzustands verwenden. Verwendet man als Funktionsargument Self, kann man damit Zustand beschreiben, der nur lokal für den aktuellen Agenten sichtbar ist. Wird Self nicht als Argument verwendet, so ist der betreffende Zustand mit anderen Agenten geteilt. Die Agenten führen ihre Interpretationsschritte in halbgeordneten Läufen aus. Dabei folgen die Schritte eines einzelnen Agenten einer vollständigen Ordnung. Die Schritte verschiedener Agenten unterliegen einer Ordnung nur dann, wenn sie sich auf gemeinsamen Zustand beziehen.

Da zur Modellierung von SDL auch das Konzept der Systemzeit definiert werden muss, wird für die verteilten Echtzeit-ASMs eine Funktion currentTime definiert, die aus Sicht jedes Agenten schwach monoton wachsend sein muss.

5.3.2 Vordefinierte Namen

Zur Definition von Domänen und Funktionen werden eine Reihe von Namen als vordefiniert angenommen. Für diese Namen wird in der formalen SDL-Semantik keine formale Definition angegeben; stattdessen wird auf die „allgemein übliche“ Definition verwiesen. Da durch die Verwendung solcher vordefinierter Namen subjektive Interpretationen möglich werden, wurde versucht, die Liste dieser Namen so klein wie möglich zu halten und nur solche Namen zu verwenden, die tatsächlich eine allgemein übliche und unbestrittene Definition besitzen. Die vordefinierten Domänen sind:

- *X* (Grundmenge der Zustandsmaschine),
- *BOOLEAN* (logische Werte),
- *NAT* (ganze Zahlen),
- *REAL* (reelle Zahlen),
- *AGENT* (Menge der Agenten),
- *PROGRAM* (Menge der Programme) und
- *TOKEN* (Terminalsymbole von Grammatiken, Zeichenketten).

Zusätzlich gibt es einige Domänenkonstruktoren: *D** ist die Domäne aller Folgen von Elementen aus *D*, *D*-**set** ist die Potenzmenge von *D*, *D1* × *D2* ist die Tupeldomäne (das Kreuzprodukt) und *D1* ∪ *D2* die Vereinigung von *D1* und *D2*.

Für diese Domänen sind eine Reihe üblicher mathematischer Funktionen definiert, darunter:

- undefined: 0-stellige statische Funktion, die ein Element aus *X* liefert, das zu keiner anderen Domäne gehört,
- für *BOOLEAN*: die 0-stelligen Funktionen *TRUE* und *FALSE*, die Vergleichsoperatoren (=, ≠) logischen Operatoren (∧, ∨, ¬, ⇔, ⇒) sowie die prädikatenlogischen Quantoren (∀, ∃, ∃!)
- für *NAT*: die Relationsoperatoren (<, ≤, >, ≥), die arithmetischen Operatoren (+, -, *, /) sowie die Literale (0, 1, ...)
- für Folgen: die leere Folge (empty), die Ermittlung des ersten und letzten Elements (head, last), die Ermittlung der Restliste (tail), die Ermittlung der Länge (length), die Bildung einer Liste aus Elementen (< v_1, ... v_n >), die Verkettung (⌢), der Elementtest (**in**), die Umwandlung in eine Menge (toSet), die Indizierung (f[n]), sowie *list comprehension* (< result | var **in** seq: cond>) und
- für Potenzmengen: die üblichen Mengenoperationen (∩, ∪, ∈, ∉, ⊂, ⊆), die leere Menge (∅), die Vereinigung einer Menge von Mengen (**U**), die Bildung einer Menge aus Elementen ({v_1, ..., v_n}), die Auswahl eines beliebigen Elements (take) sowie *set comprehension* ({result | var ∈ set: cond}).

Zur Notation von Ausdrücken stehen zusätzlich die folgenden Konstrukte zur Formulierung von Ausdrücken bereit:

- der Funktionsruf einer mehrstelligen Funktion ($f(v_1, ..., v_n)$),
- der Aufruf einer einstelligen Funktion (v.f),
- der Aufruf einer nullstelligen Funktion (f),
- der bedingte Ausdruck (**if** bedingung **then** term1 [**elseif** bedingung **then** term2] [**else** term3] **fi**),
- die Auswahl eines Ausdrucks auf Grund eines Musters:

```
case term of
    | muster1: term1
    | muster2: term2
    ...
    [otherwise: term0]
endcase
```

- die Konstruktion eines Tupels (**mk**-*D*(v_1, ... v_n)) und
- die Projektion eines Elements von einem Tupel (**s-***D*(v)).

5.3.3 Definition von Grammatiken

Grammatiken werden mit einer speziellen Syntax notiert. Diese Notation ist jedoch lediglich eine Kurzschreibweise für die Definition von weiteren Tupeldomänen und Vereinigungen.

Die Syntax für Grammatikdefinitionen folgt der BNF. Jedoch dürfen Alternativen jeweils nur ein Symbol enthalten, sie definieren Vereinigungen. Sind mehrere Symbole auf der rechten Seite, so definiert die Grammatikregel eine Tupeldomäne. Geschweifte Klammern zeigen Gruppierung an, eckige Klammern zeigen optionale Teile an. Der Kleene-Stern (*) bedeutet eine Folge von Elementen, der Suffix **-set** eine (ungeordnete) Menge. In Tupel-Regeln werden linke und rechte Seite durch Doppelpunkte (::), in Vereinigungsregeln durch Gleichheitszeichen getrennt.

Beispiel 10. Die Regel

$Symbol = Symbol_1 \mid Symbol_2 \mid ... \mid Symbol_n$

ist gleichbedeutend mit

$Symbol =_{def} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$

Das Symbol $=_{def}$ definiert dabei einen neuen abgeleiteten Namen. Die Regel

$Symbol :: Symbol_1\ \ Symbol_2{+}\ Symbol_3$**-set** $[Symbol_4]\ Symbol_5{*}$

ist fast gleichbedeutend[12] mit

$Symbol =_{def} Symbol_1 \times Symbol_2{*} \times Symbol_3$**-set** $\times Symbol_4 \times Symbol_5{*}$

Für die Elemente der Domäne *Symbol* gibt es dann Zugriffsfunktionen (*selector functions*) **s-***Symbol*$_1$, **s-***Symbol*$_2$**-seq**, **s-***Symbol*$_3$**-set**, **s-***Symbol*$_4$ und **s-***Symbol*$_5$**-seq**. **s-***Symbol*$_4$ liefert undefined, wenn das optionale Symbol nicht vorhanden ist.

12. Tatsächlich führt diese Grammatikdefinition eine weitere Domäne im Modus **controlled** ein, mit der die Identität von Knoten im abstrakten Syntaxbaum repräsentiert wird. Den Elementen dieser Domäne sind dann die im Beispiel definierten Tupel zugeordnet.

5.4 Teil 2: Statische Semantik

Eine Übersicht über die Definitionen der statischen Semantik gibt Abbildung 9.

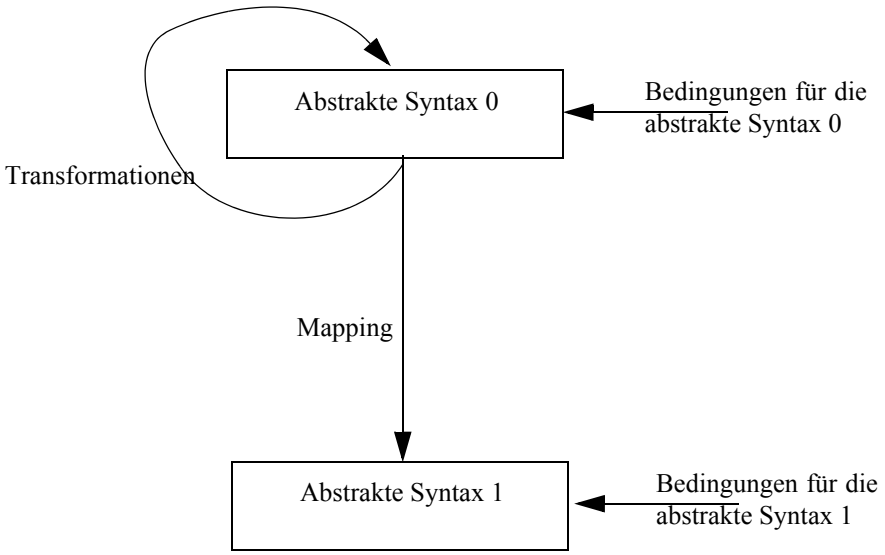

Abbildung 9: Statische Semantik von SDL

Für jedes Konzept von SDL finden sich in statischen Semantikdefinition folgende Definitionen:

- Definition der abstrakten Syntax 0 (AS0),
- Bedingungen für die abstrakte Syntax 0,
- Transformationsregeln,
- Definition der abstrakten Syntax 1 (AS1),
- Definition eines Fragments[13] der Mapping-Funktion von AS0 auf AS1,
- Bedingungen der abstrakten Syntax 1 und
- Hilfsfunktionen, die für die anderen Teile benötigt werden.

Die abstrakte Syntax 0 ist der Ausgangspunkt der formalen Semantikdefinition. Sie ist in ihrer Definition der konkreten Syntax so nahe, dass die Abbildung von konkreter (textueller) Grammatik auf die abstrakte Syntax 0 als offensichtlich gilt. Für die abstrakte Syntax 0 werden eine Reihe von Transformationen ausgeführt. Schließlich wird sie auf die abstrakte Syntax 1 durch die Mapping-Funktion abgebildet. Für AS0 und AS1 sind prädikatenlogische Bedingungen definiert, die eine SDL-Spezifikation erfüllen muss, damit sie als richtig (*well-formed*) gilt. Die abstrakte Syntax 1 ist Ausgangspunkt der dynamischen Semantik.

5.5 Teil 3: Dynamische Semantik

Die Information, die ein Baum der abstrakten Syntax 1 enthält, kann man grob in die Struktur, das Verhalten, und die Daten eines SDL-Systems gliedern. Diese Informationen müssen in der dynamischen Semantik durch Programme von ASM-Agenten repräsentiert werden.

Dazu liegt es nahe, SDL-Agenten auf ASM-Agenten abzubilden. Es zeigt sich, dass zusätzlich Agenteninstanzmengen und Kanäle (Signalfluss) durch ASM-Agenten repräsentiert wer-

13. Die Mapping-Funktion ist insgesamt durch einen case-Ausdruck definiert. Jedes Fragment definiert eine Reihe von Mustern in diesem Ausdruck.

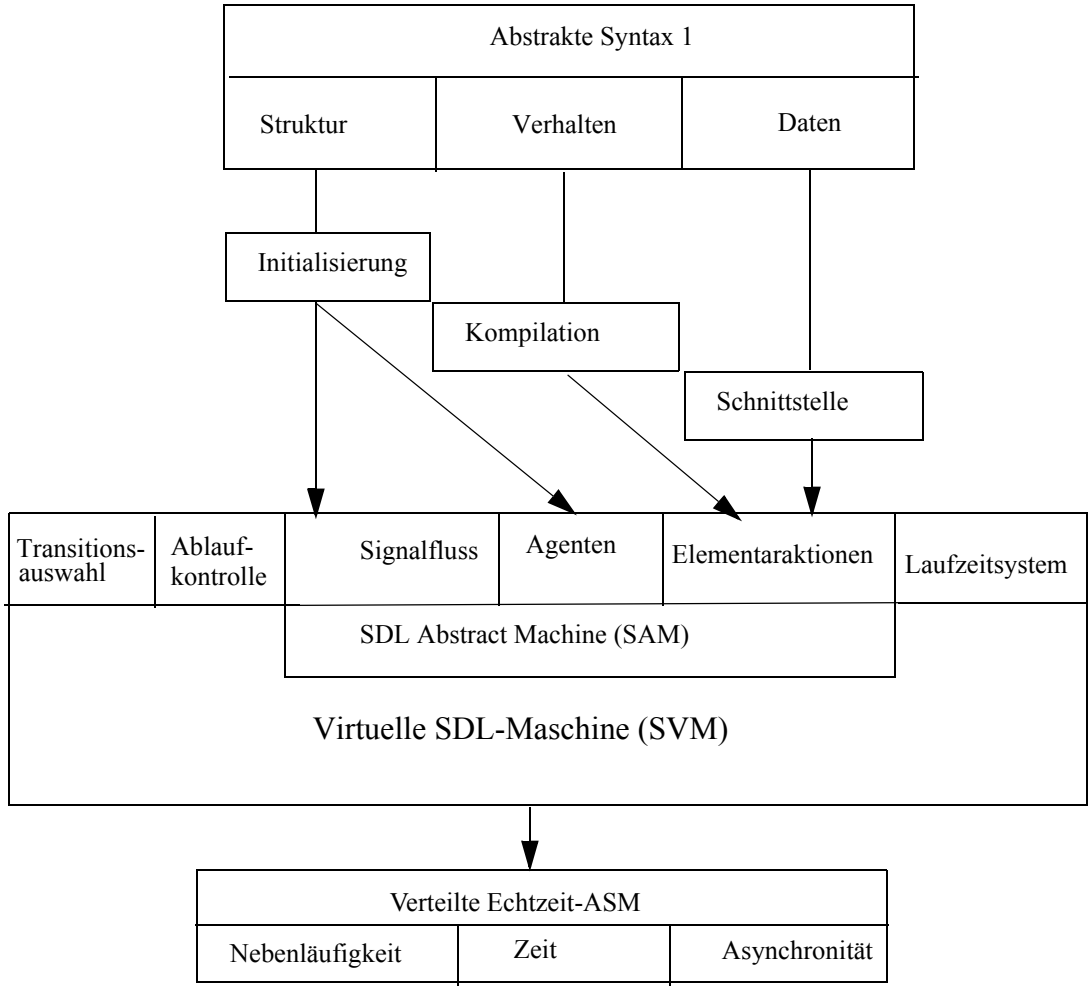

Abbildung 10: Dynamische Semantik von SDL

den müssen. Die Erzeugung dieser Agenten erfolgt in der Initialisierungsphase der Interpretation. Dazu traversiert ein Initialisierungsprogramm die in der abstrakten Syntax dargestellte Struktur und erzeugt die initialen ASM-Agenten. Diejenigen ASM-Agenten, die SDL-Agenten repräsentieren, beginnen anschließend mit der Abarbeitung der Starttransitionen.

Abbildung 10 stellt eine Übersicht über die dynamische Semantik von SDL dar. Hauptinhalt der dynamischen Semantik ist die virtuelle SDL-Maschine (*SDL virtual machine,* SVM), die auf Basis der abstrakten Syntax das Verhalten eines SDL-Systems in Form von ASM-Agenten repräsentiert. Ein Teil dieser virtuellen SDL-Maschine besteht aus Funktionen und ASM-Programmen, die unabhängig von der konkreten SDL-Spezifikation das SDL-Laufzeitsystem bilden. Dazu gehören neben zahlreichen Hilfsfunktionen insbesondere die Algorithmen zur Transitionsauswahl (*selection*) und Ablaufkontrolle (*firing*). Die Abbildung der abstrakten Syntax auf ASM-Agenten benutzt direkt nur einen Teil der SVM, der abstrakte SDL-Maschine (*SDL abstract machine*, SAM) genannt wird.

Die Transitionen eines SDL-Agenten werden in Elementaraktionen (*primitives*) der SAM übersetzt. Diese Übersetzung erfolgt durch eine Kompilationsfunktion.

In manchen Aktionen der SDL-Agenten wird auf Datentypen Bezug genommen. Das Datentypsystem wird in die formale Semantik durch eine funktionale Schnittstelle integriert (siehe Abschnitt 8.1). Die Elementaraktionen verwenden Funktionen dieser Schnittstelle, die ihrerseits die datentyp-relevanten Knoten der abstrakten Syntax 1 verwenden.

Die virtuelle SDL-Maschine wird durch eine Reihe von ASM-Programmen definiert, die von einer verteilten Echtzeit-ASM abgearbeitet werden. Die möglichen Abläufe dieser Maschine stellen die Semantik des SDL-Systems dar.

5.6 Fazit

Die formale Spachdefinition von SDL besteht aus drei Teilen: Einleitung, statischer und dynamischer Semantik. Alle drei Teile basieren auf dem ASM-Kalkül, auf dessen Grundlage Abstraktionen definiert wurden, die für jeden Teil der Sprachdefinition eine leichter verständliche und kompaktere Notation der Semantik erlauben.

6 Die Datentypen von SDL-2000

Gegenüber SDL-92 wurde das Datentypsystem in SDL-2000 vollständig überarbeitet. Das neue Datentypmodell ist gegenüber dem alten vor allem durch die folgenden Eigenschaften charakterisiert:

- Neben der Wertesemantik bietet SDL nun auch eine Referenzsemantik, die ihrerseits Grundlage eines objektorientierten Datentypsystems ist.
- Die Semantik eines Datentyps ergibt sich nicht mehr durch die Definition von Axiomen über Terme von Operatoranwendungen, sondern einerseits durch Konstruktion neuer Datentypen aus Elementartypen, und andererseits durch die algorithmische Definition von Operatoren und Methoden.
- Die Konstruktionsverfahren für neue Datentypen wurden so ausgewählt, dass eine geradlinige Abbildung von ASN.1 [X.680] auf SDL im Rahmen von [Z.105] möglich wird.

Diese Änderungen hatten weit reichende Auswirkungen sowohl auf die gesamte SDL-Sprachdefinition und die Formalisierung des Datentypkalküls. Verglichen mit SDL-92 sind die vielleicht einschneidendsten Auswirkungen die folgenden:

- Durch Ersetzen der algebraischen Definition von Datentyp auf Basis von Axiomen durch algorithmische Definitionen entfällt das in [Sch02] identifizierte Problem der prinzipiellen Unberechenbarkeit von Datentypen; Datentypen werden damit auch effizient implementierbar (siehe auch Abschnitt 3.4). Der Verzicht auf ein separates Kalkül erlaubt es, eine einheitliche semantische Grundlage für die gesamte SDL-Semantik zu finden.
- Durch die Einführung von identifizierbaren Objekten und die Möglichkeit zur Definition von Methoden an diesen Objekten ist die Berechnung eines Ausdrucks nun nicht mehr notwendigerweise seiteneffektfrei. Da diese Änderung einen deutlichen Bruch mit der SDL-Tradition darstellt, wurden von den Autoren des Sprachstandards Versuche unternommen, Seiteneffekte in Operationen (Operatoren und Methoden) in bestimmten Kontexten zu untersagen. Letztlich hat sich aber gezeigt, dass eine solche Einschränkung entweder nicht überprüfbar ist oder aber die Sprache so stark einschränkt, dass sie unbenutzbar wird (siehe Abschnitt 6.6).
- Da Operationen nun oft durch Algorithmen definiert werden, unterliegt die Abarbeitung von Operationen einem Zeitverbrauch. Insbesondere ist es möglich, dass die Berechnung eines Operators nicht terminiert.
- Durch die Einführung von virtuellen Operationen ist es statisch nicht mehr möglich anzugeben, welcher Operationsruf durch welche Operationsdefinition implementiert wird.
- Mit der Definition einer Referenzsemantik und der Möglichkeit, rekursive und zyklische Datenstrukturen zu definieren, ergibt sich praktisch die Frage der Speicherverwaltung[14]. Da eine explizite Speicherverwaltung durch das SDL-Programm einen weiteren Bruch mit der SDL-Tradition dargestellt hätte, ist die Lebenszeit von Objekten in SDL-2000 unspezifiziert – es ist Aufgabe der SDL-Implementierung, den Speicher nicht verwendeter Objekte freizugeben.

Dieses Kapitel erläutert alle Datentypkonzepte von SDL-2000, um den Leser dieser Arbeit in die Lage zu versetzen, die in den folgenden Kapiteln angegebenen Beispiel für die Definition der formalen Semantik in das Gesamtbild der SDL-Konzepte einzuordnen. Die Vorstellung der Datentypen orientiert sich dabei an der abstrakten Syntax 0, die einerseits sich stark an die konkrete Syntax anlehnt, andererseits aber unmittelbare Basis vieler Formeln der statischen Seman-

14. Die Wertesemantik von SDL-92 legte eine Implementierungsstrategie nahe, die keine Kontrolle der Speicherverwaltung durch das SDL-Programm erforderte.

tikdefinition ist. Auch wenn der Leser damit die konkrete Syntax hier nicht findet, so kann er doch anhand der Beispiele und erläuternden Text einen Eindruck der konkreten Syntax gewinnen.

6.1 Datentypdefinitionen

Eine Datendefinition (<data definition>) definiert zum einen eine Menge möglicher Werte des Datentyps (die *Sorte*[15], engl. *sort*), zum anderen eine Menge von Operationen auf der Sorte. Es gibt mehrere Arten von Datendefinitionen:

<data definition> =
    <data type definition> | <interface definition> | <syntype definition> | <synonym definition>

Hierbei werden nur mit Hilfe einer <data type definition> tatsächlich neue Sorten eingeführt. Eine <interface definition> wählt aus der Menge aller Pid-Werte eine Teilmenge aus (nämlich die Menge aller der Agenten, die diese Schnittstelle besitzen). Pid-Sorten werden also nicht direkt durch eine Typdefinition eingeführt, sondern als Seiteneffekt einer <interface definition>. Eine <syntype definition> definiert einen Aliasnamen für einen existierenden Datentyp, und schränkt eventuell die Menge der gültigen Werte ein.

Neben den Pid-Sorten gibt es zwei verschiedene Kategorien von Sorten: Die Objektsorten, deren Werte als Referenzen übergeben werden, und die Wertesorten, die einer Kopiersemantik unterliegen.

Eine <synonym definition> definiert keinen Datentyp, sondern eine symbolische Konstante (ein Synonym). <synonym definition> ist nur deshalb eine Alternative für <data definition>, weil Synonyme syntaktisch an allen Stellen erlaubt sind, an denen auch alle anderen Datendefinitionen erlaubt sind.

Jede Datendefinition trägt einen Namen. Die von der Datendefinition eingeführte oder eingeschränkte Sorte trägt den gleichen Namen. Mit dem Konstrukt <sort> kann man auf eine Sorte Bezug nehmen:

<sort> = <basic sort> | <anchored sort> | <expanded sort> | <reference sort> | <pid sort>

Üblicherweise erfolgt eine solche Bezugnahme durch den Namen der Sorte. Beispielsweise wird in der Deklaration

```
dcl counter Integer;
```

Bezug auf die (vordefinierte) Sorte Integer genommen. In diesen Fällen handelt es sich um die Regel <basic sort>:

<basic sort> = <sort identifier>[16] | <syntype>

Unter Umständen kann der Name einer Sorte, die beispielsweise als Rückgabetyp einer Operation auftritt, nicht angegeben werden, weil sich der Rückgabetyp der Operation bei Spezialisierung des Datentyps ändert. In diesem Fall kann man mit dem Schlüsselwort **this** die Sorte flexibel angeben. Mit der Spezialisierung des Datentyps ändert sich auch automatisch jedes Auftreten dieses Sortennamens:

```
object type Base;
```

15. Der Begriff der Sorte wird üblicherweise im Zusammenhang mit abstrakten Datentypen verwendet, um die mathematischen Objekte zu bezeichnen, die als Argumente und Ergebnis einer Operationssignatur erscheinen. Im streng algebraischen Sinne sind Sorten nicht selber Mengen, sondern symbolische Namen, denen eine Trägermenge zugeordnet ist.
16. Der unterstrichene Teil eines Hilffsymbols der Grammatik bezeichnet eine sogenannte „semantische Kategorie“. Syntaktisch erlaubt die Regel <basic sort> die Angabe beliebiger <identifier>. Für diese wird aber durch die semantische Kategorie gefordert, dass es sich um Sortenbezeichner handelt.

```
    methods
        merge(this) -> this;
endobject type;

object type Derived inherits Base;
endobject type;
```

In diesem Beispiel definiert der Typ Base eine Methode merge, die ein Argument der gleichen Sorte erwartet und ein ebensolches Ergebnis liefert. Der Typ Derived ist eine Spezialisierung des Typs Base. Wird nun die Methode merge für ein Objekt der Sorte Base aufgerufen, so ist der Rückgabewert ebenfalls aus der Sorte Base. Erfolgt der Aufruf jedoch für ein Objekt der Sorte Derived, so muss das Argument ebenfalls aus der Sorte Derived sein, und auch das Ergebnis ist von der Sorte Derived. In der abstrakten Syntax wird eine solche Sortenreferenz als <anchored sort> bezeichnet. Die Referenz enthält optional den Namen der Basissorte:

```
<anchored sort> :: [<basic sort>]
```

Wenn eine Sorte als Objektsorte definiert wurde (wie in obigem Beispiel die Sorten Base und Derived), dann werden die Werte der Sorte als Referenzen bei Zuweisungen und als Argument von Operatoren übergeben. Es ist jedoch möglich, an ausgewählten Stellen eine Übergabe per Wert festzulegen. Dazu schreibt man vor den Sortennamen das Schlüsselwort **value**:

```
dcl base_value value Base;
```

Zuweisungen von und an die Variable base_value erfolgen nun durch Kopienbildung. Eine solche Konstruktion wird in der abstrakten Syntax 0 als <expanded sort> repräsentiert:

```
<expanded sort> :: <basic sort>
```

Genauso ist es möglich, für eine durch Kopiersemantik definierte Sorte Referenzsemantik festzulegen. Dies erfolgt analog durch das Schlüsselwort **object**. So definierte Variablen und Parameter werden bei Belegung mit einer Referenz initialisiert. Zugriffe liefern dementsprechend Referenzen. Dieses Konstrukt wird in der abstrakten Syntax als <reference sort> repräsentiert:

```
<reference sort> :: <basic sort>
```

Schließlich kann sich ein Sortenname auch auf eine Pid-Sorte beziehen:

```
<pid sort> = <sort<identifier>
```

In diesem Fall ergibt sich die Information, dass es sich um eine Pid-Sorte handelt, allein durch den Bezeichner (<identifier>): dieser muss eine Schnittstellendefinition bezeichnen. In den anderen Fällen (<anchored sort>, <expanded sort> und <reference sort>) ergibt sich die Art der Sorte nicht allein aus dem enthaltenen Sortenbezeichner, da die jeweilige Konstruktion ja gerade eine Modifikation der Sorte vornimmt. Deshalb wird also das Hilfssymbol <pid sort> als Synonym für <identifier> definiert, während die Symbole <anchored sort>, <expanded sort> und <reference sort> eigene Knoten der abstrakten Syntax bilden.

Eigentliche Datentypdefinitionen werden durch die Konstrukte **object type** und **value type** definiert. Die abstrakte Syntax für Datentypdefinitionen lautet

```
<data type definition> ::
    <package use clause>* <type preamble> <data type heading> [<data type specialization>]
    [ <data type definition body> ]
```

Mit Hilfe der Liste <package use clause>* kann man Pakete referenzieren, auf deren Inhalt in der Datentypdefinition Bezug genommen werden kann. Das Konstrukt <type preamble> kann eines der Schlüsselwörter **virtual** oder **abstract** enthalten. Dementsprechend ist der entstehende Datentyp virtuell oder abstrakt. Ein virtueller Datentyp ist ein Typ, der in einer Spezialisierung seines Containers (also beispielsweise des Prozesstyps, in dem er definiert ist) durch

einen anderen Typ ausgetauscht werden kann. Ein abstrakter Datentyp ist ein Typ, von dem keine Instanzen gebildet werden können.

Das Konstrukt <data type heading> gibt die Art der Datentypdefinition (Objekttyp oder Wertetyp), den Namen, eventuelle Kontextparameter und Einschränkungen bei der Redefinition eines virtuellen Typs an:

<data type heading> ::
    <data type kind> <data type><name> <formal context parameter>* [<virtuality constraint>]

<data type kind> = **value** | **object**

Durch Angabe von Kontextparametern erhält man parametrisierte Typen. Kontextparameter können beispielsweise weitere Datentypen sein, die innerhalb des gerade definierten Datentyps als Feldtypen auftreten. Diese Kontextparameter können bei Spezialisierung gebunden werden. Die aktuellen Kontextparameter geben dann den tatsächlichen Feldtyp an. Darauf wird im Abschnitt 6.3 näher eingegangen.

Der eigentliche Körper der Datentypdefinition wird durch die Regel <data type definition body beschreiben>:

<data type definition body> ::
    <entity in data type>* [<data type constructor>] <operations> [<default initialization>]

In einer Datentypdefinition können weitere Definitionen (sog. *entities*) enthalten sein. Welche diese genau sind, wird durch die Regel <entity in data type> beschrieben

<entity in data type> =
    <data type definition> | <syntype definition> | <synonym definition> | <exception definition>

Da Daten in SDL passiv sind, ist die Definition aktiver Komponenten (also von Agenten) in Datentypen nicht erlaubt; erlaubt sind lediglich Definitionen weiterer Daten und Ausnahmen.

Den enthaltenen Definition folgt optional ein Typkonstruktor (<data type constructor>), der in Abschnitt 6.5 erläutert wird. Er gibt an, auf welche Weise dieser Datentyp aus anderen konstruiert ist – es handelt sich also nicht um Konstruktoren für Instanzen, wie sie aus anderen Programmiersprachen bekannt sind.

Dem Typkonstruktor folgt die Deklaration von Operationen (Operatoren und Methoden). Dabei werden zuerst die Signaturen angegeben, gefolgt von den Definitionen:

<operations> :: <operation signatures> <textual operation reference>* <operation definitions>*

Falls einige Operationen nicht textuell in der Datentypdefinition definiert sind, so muss in der Typdefinition zumindest eine Operationsreferenz (<textual operation reference>) auftauchen.

Optional kann eine Datentypdefinition eine Defaultinitialisierung (<default initialization>) enthalten. Diese Initialisierung wird immer dann als Initialwert verwendet, wenn eine Variable ohne speziellen Initialwert definiert wurde:

<default initialization> ::[<virtuality>] [<constant expression>]

Beispiel 11. Eine Datentypdefinition, die von allen Konstrukten Gebrauch macht, ist folgender Objekttyp DT:

```
use TypesLibrary/Color;  /* package use clause */
virtual object type DT<type P>  /* type preamble, data type kind, name, formal context parameter */
    atleast BT1                  /* virtuality constraint */
    inherits BT2;                /* specialization */
    exception                    /* entity in data type */
        InvalidColor(Color);
    struct                       /* data type constructor
        private field1 Color;
```

```
    optional field2 P;
  methods                          /* operation signatures */
    getP() -> P;
    setColor(Color) raises InvalidColor;
  method setColor; referenced;  /* textual operation reference */
  method getP()                    /* operation definition */
  {
    return field2;
  }
  default (. Red .);                /* Default initialization */
endobject type;
```

In diesem Typ ist DT eine Spezialisierung von BT2. Redefinitionen von DT müssen nicht notwendigerweise auch Spezialisierungen von DT sein – es reicht, wenn sie Spezialisierungen von BT1 sind. Daraus ergibt sich implizit, dass BT2 eine Spezialisierung von BT1 ist; anderenfalls würde ja DT seine eigene Virtualitätsbedingung verletzen.

6.2 Interface-Definitionen

Eine Schnittstelle (*interface*) legt die Menge von Signalen fest, über die mit einem Agenten oder über einen Kanal kommuniziert werden kann. Dies kann eine Teilmenge aller Signale sein, die der Agent senden und empfangen kann. Jede Agententypdefinition impliziert eine Schnittstellendefinition, die nämlich die Menge aller Signale umfasst, die an den Gates dieses Agenten ausgetauscht werden kann[17].

In diesem Zusammenhang bezeichnet „Signal“ nicht nur die direkt deklarierten Signale, sondern auch diejenigen, die aus der Deklaration von entfernten Prozeduren und Variablen im Rahmen des Transformationsprozesses entstehen. In der Schnittstellendefinition können dementsprechend im Allgemeinen nicht nur Signale, sondern auch Variablen und Prozeduren angegeben werden.

Schnittstellen stehen in einer Spezialisierungshierarchie. Abweichend von anderen Spezialisierungen ist bei Schnittstellen Mehrfachvererbung möglich: Die Signalmenge einer Schnittstelle ergibt sich aus der Vereinigung der Signalmengen der Basisschnittstellen zuzüglich der direkt in der Schnittstelle aufgeführten Signale. Bei Vererbung von Agententypen entsteht die implizite Schnittstelle des spezialisierten Agenten aus der impliziten Schnittstelle des Basisagenten, sowie aller Schnittstellen, die an Gates in Richtung zum Agenten aufgeführt sind.

Ein Signal kann in einer Schnittstelle definiert werden. Ist die Definition des Signals bereits woanders erfolgt, kann das Signal in der Schnittstelle referenziert werden.

Eine Schnittstellendefinition impliziert die Definition einer Pid-Sorte. Pid-Sorten sind Spezialisierungen der Sorte Pid, also der Sorte, deren Werte Referenzen auf Agenten sind. Während alle Referenzen auf Agenten Element der Sorte Pid sind, sind in einer Pid-Sorte, die einer Schnittstelle S entspricht, nur Referenzen auf Agenten enthalten, deren implizite Schnittstelle eine Spezialisierung von S ist. Damit wird in SDL typsichere Kommunikation möglich: Beim Senden eines Signals kann schon festgestellt werden, ob der empfangende Agent das Signal auch verarbeiten kann (wobei das Verwerfen des Signals in einem Zustand hier auch als Verarbeitung gilt).

Eine Schnittstellendefinition erfolgt durch die Regel <interface definition>:

```
<interface definition> ::
      <package use clause>* [<virtuality>] <interface heading> [<interface specialization>]
      <entity in interface>* <interface use list>
```

17. Tatsächlich impliziert eine Agentdefinition mehrere Schnittstellendefinitionen, nämlich je eine für die Zustandsmaschine des Agenten, für den Typ des Agenten, und für den Agenten selbst.

Wie auch in Datentypdefinitionen kann eine Schnittstellendefinition mit einer Liste verwendeter Pakete beginnen, optional gefolgt vom einem der Schlüsselwort **virtual**, **redefined** oder **finalized**. Die Angabe des Schlüsselworts **abstract** ist hier nicht erlaubt, da Schnittstellen sowieso nicht instanziert werden können, sondern neue Werte immer nur durch Erzeugung von Agenten entstehen.

Der Namen der Schnittstelle, die formalen Kontextparameter und Einschränkungen der Redefinition virtueller Schnittstellen finden sich in der Regel für den Schnittstellenkopf (<interface heading>):

```
<interface heading> ::
    <interface<name> <formal context parameter>* [<virtuality constraint>]
```

Auf die Angabe des Schnittstellenkopfs folgt optional die Festlegung von Basisschnittstellen; dies ist in Abschnitt 6.3 erläutert. Darauf folgt die Definition von Elementen, die in der Schnittstellendefinition enthalten sind. Diese sind durch die Regel <entity in interface> festgelegt:

```
<entity in interface> =
        <signal definition>
    |   <interface variable definition>
    |   <interface procedure definition>
    |   <exception definition>
```

Wie bereits erläutert, kann eine Schnittstellendefinition die Definition von Signalen, entfernten Variablen und entfernten Prozeduren enthalten. Zusätzlich ist die Definition von Ausnahmen erlaubt, die dann beispielsweise in der Definition entfernter Prozeduren verwendet werden können.

Soll auf bereits definierte Signale zurückgegriffen werden, erfolgt dies durch die Regel <interface use list>:

```
<interface use list> :: <signal list item>*
```

Die hier nicht angegebene Regel <signal list item> erlaubt den Verweis auf Signale, Signallisten, Timer, entfernte Prozeduren, Schnittstellen und entfernte Variablen.

Die Definition von entfernten Variablen und Prozeduren erfolgt innerhalb von Schnittstellen syntaktisch etwas anders als außerhalb, da in der Schnittstelle das Schlüsselwort remote nicht erforderlich und deshalb nicht erlaubt worden ist:

```
<interface variable definition> :: <remote variable<name>+ <sort>

<interface procedure definition> :: <remote procedure<name> <procedure signature>
```

Beispiel 12. Die Schnittstelle, die der Abbildung der OMG-IDL-Schnittstelle CosNaming::NamingContextExt [OMG02b] folgt, wie sie in [Z.130] definiert ist, lautet in SDL

```
use CORBA;
interface NamingContextExt inherits NamingContext
{
    exception InvalidAddress;
    resolve_str(Charstring) -> Object
        raises InvalidName;
    to_name(Charstring) -> Name
        raises InvalidName;
    to_string(Name) -> Charstring
        raises InvalidName, InvalidAddress;
    to_url(Name) -> Charstring
        raises InvalidName, InvalidAddress;
}
```

6.3 Spezialisierung

Die Spezialisierung von Datentypen in SDL stellt wie die Spezialisierung von Agenten eine Vererbungsrelation dar. Der spezialisierte Typ verfügt über alle Eigenschaften (enthaltene Variablen, Typen, Transitionen usw.) des Basistyps und kann zusätzlich weitere Eigenschaften definieren.

Die Spezialisierung von Typen impliziert nicht automatisch, dass der spezialisierte Typ polymorph im Sinne des Liskovschen Substitutionsprinzips [Z.100-00] verwendet werden kann. Statt dessen sind polymorphe Zuweisungen im Allgemeinen nur zwischen Objektsorten und Pid-Sorten möglich; bei Wertesorten ist Typgleichheit erforderlich. Genauer werden die Zuweisungsregeln durch die Begriffe *typverträglich* und *unmittelbar typverträglich* definiert [Z.100-00, 12.1.3]:

Seien T und V Sorten. V ist typverträglich mit T genau dann, wenn:

a) V und T dieselbe Sorte sind,

b) V unmittelbar typverträglich mit T ist, oder

c) T eine Objekt- oder Pid-Sorte ist und es eine Sorte U gibt, so dass V typverträglich mit U und U typverträglich mit T ist.

V ist unmittelbar typverträglich mit T, wenn

a) V durch die Regel <basic sort> beschrieben wird und T eine Basissorte von V ist,

b) V durch die Regel <anchored sort> beschrieben wird und T die dort enthaltene Sorte ist,

c) V durch die Regel <reference sort> beschrieben wird und T die dort enthaltene Sorte ist,

d) T durch die Regel <reference sort> beschrieben wird und V die dort enthaltene Sorte ist,

e) V durch die Regel <expanded sort> beschrieben wird und T die dort enthaltene Sorte ist,

f) T durch die Regel <expanded sort> beschrieben wird und V die dort enthaltene Sorte ist,

g) V eine Pid-Sorte und T eine Basissorte von V ist.

Der Begriff der Typverträglichkeit, der eine reflexive Halbordnung definiert, wird seinerseits zur Definition der folgenden Konstrukte verwendet:

- Typverträglichkeit von Operationen (6.4) und
- Zuweisungen (6.11).

Durch die Definition von Objekttypen wird es möglich, Variablen mit polymorphen Werten zu definieren. Eine solche polymorphe Variable hat einen statischen Typ und einen dynamischen Typ.

Beispiel 13. Gegeben seien folgende Definitionen:

```
object type Base{
    struct
        x Integer;
}
object type Derived inherits Base{
    struct
        y Integer;
}
dcl var Base := <<type Derived>>(. 3, 4 .);
```

Der statische Typ der Variablen var ist Base; der dynamische Typ ist Derived.

Die Spezialisierung von Datentypen wird durch die Regel <data type specialization> festgelegt:

<data type specialization> :: <data type<type expression> <renaming>

Innerhalb der Spezialisierung wird durch den Knoten <type expression> der Basistyp angegeben. Dieser kann lediglich der Typname sein oder aber ein parametrisierter Basistyp, an den aktuelle Kontextparameter übergeben werden. Im folgenden Beispiel wird der bereits definierte parametrisierte Typ DT spezialisiert und dabei festgelegt, dass der formale Kontextparameter durch den Typ Integer aktualisiert wird.

```
object type DT2 inherits DT<Integer>;
endobject type;
```

Bei der Spezialisierung von Typen können Operationsnamen geändert werden. Dies erfolgt durch die Regel <renaming>, die ihrerseits aus einer Folge von <rename pair>-Konstrukten besteht:

<renaming> :: <rename pair>*

<rename pair> =
    <rename pair gen operation name> | <rename pair gen literal name>

Umbenennungen können sowohl Operationsnamen als auch Literalnamen (bei Aufzählungstypen ändern):

<rename pair gen operation name> :: <operation name> <base type<operation name>

<rename pair gen literal name> :: <literal name> <base type<literal name>

Beispielsweise könnte bei Spezialisierung des Typs DT die Operation getP umbenannt werden:

```
object type DT3 inherits DT<Integer>(getInt = getP);
endobject type;
```

Die Syntaxregel für Schnittstellenspezialisierung unterscheidet sich von der Datentypspezialisierung dadurch, dass bei Schnittstellen Mehrfachvererbung erlaubt ist:

<interface specialization> :: <interface<type expression>+

In der konkreten Syntax werden die Basisschnittstellen durch Kommata getrennt:

```
interface Derived inherits Base1, Base2;
endinterface;
```

6.4 Operationen

Durch Operationen wird das Verhalten eines Datentyps festgelegt. Es gibt zwei Arten von Operationen: Operatoren und Methoden. Beide können durch die Regel <operation signatures> deklariert werden, die Definition des Verhaltens einer Operation wird in Abschnitt 6.6 erläutert.

Durch Operatoren werden statische Operationen des Datentyps definiert: Bei einem Operatorruf ergibt sich der gerufene Operator allein aus dem Operatornamen und den statischen Argumenttypen. Durch Methoden können auch dynamische Operationen definiert werden, hier ergibt sich die gerufene Operation erst aus den dynamischen Argumenttypen.

<operation signatures> :: <operator list> <method list>

<operator list> ::<operation signature>*

<method list> ::<operation signature>*

In der konkreten Syntax werden die Operatorsignaturen mit dem Schlüsselwort **operators** eingeleitet und die Methodensignaturen mit dem Schlüsselwort **methods**. Beide Arten von Operationen werden durch eine <operation signature> deklariert. Während diese Signatur für

Operatoren sämtliche Parametertypen angibt, gibt es bei Methoden einen zusätzlichen impliziten Parameter, auf den innerhalb der Methodendefinition durch das Schlüsselwort **this** zugegriffen werden kann. Die Komponenten einer <operation signature> lauten

<operation signature> ::
    <operation preamble> { <operation name> | <name class operation> }
    <argument>* [<result>] <raises>

Mit Hilfe der <operation preamble> kann festgelegt werden, ob die Operation virtuell ist (ob es sich also um eine dynamische Operation handelt). Diese Angabe ist nur für Methoden erlaubt. Außerdem kann die Sichtbarkeit der Methode bestimmt werden (siehe Abschnitt 6.8)

<operation preamble> :: [<virtuality>] [<visibility>]

Der Name der Operation kann entweder als <operation name> oder als <name class operation> angegeben werden, letzteres Konstrukt wird in Abschnitt 6.7 vorgestellt. Ein <operation name> ist entweder ein Bezeichner oder ein <quoted operation name> (also eine Zeichenkette in doppelten Anführungszeichen). Mit Hilfe des Konstrukts <quoted operation name> können Operationen wie **"+"** oder **"and"** definiert werden.

<operation name> =<name> | <quoted operation name>

Auf den Namen folgt die Liste der Argumenttypen. Für jeden Parameter kann angegeben werden, ob er ein virtueller Parameter ist und von welcher *Übergabeart* der Parameter ist. Außerdem muss natürlich der Parametertyp festgelegt werden:

<argument> ::[<argument virtuality>] <formal parameter>

<formal parameter> :: <parameter kind> <sort>

Die Angabe <argument virtuality> ist nur für virtuelle Methoden erlaubt, sie bedeutet, dass dieser Parameter bei der dynamischen Auswahl der Methode berücksichtigt wird (siehe Abschnitt 6.11).

<argument virtuality> ::

Durch die Übergabeart kann angegeben werden, ob der Parameter Eingabeparameter, Ausgabeparameter oder beides ist. Parameter der Arten **out** und **inout** führen dazu, dass innerhalb der Operation die Variable, die als Argument übergeben wurde, geändert werden kann. Falls die Übergabeart nicht angegeben wurde, so wird implizit **in** verwendet.

<parameter kind> =[**inout** | **in** | **out**]

Optional kann eine Operation ein Ergebnis liefern und Ausnahmen auslösen, dies wird durch die Konstrukte <result> und <raises> deklariert.

<result> :: <sort>

<raises> = <exception><identifier>*

Für Signaturen wird der Begriff der Typverträglichkeit wie folgt erweitert:

Eine Operationssignatur S1 ist typverträglich mit einer Operationssignatur S2, wenn

a) S1 und S2 die gleiche Zahl von Argumenten haben,

b) für jedes virtuelle Argument A1 von S1 ist das entsprechende Argument A2 von S2 ebenfalls virtuell und A1 ist typverträglich mit A2 und

c) für jedes nicht-virtuelle Argument A1 von S1 ist das entsprechende Argument A2 von S2 ebenfalls nicht-virtuell und A1 hat die gleiche Sorte wie A2

Für jeden Ruf einer dynamischen Methode wird die Methodendefinition durch den Namen sowie die dynamischen Typen der Argumente bestimmt. Kandidatenmethoden sind dann diejenigen Methoden M, für die die dynamischen Argumente typverträglich mit den Argumentsorten von M sind. Falls es mehrere solcher Methoden gibt, wird diejenige ausgewählt, die „am speziellsten“ ist. Dazu ist es nötig sicherzustellen, dass es stets genau eine solche „speziellste“ Methode gibt, dies wird durch folgende Forderung sichergestellt:

Für jede Methode wird eine dynamische Signatur gebildet. Diese dynamische Signatur besteht aus der Menge aller Methoden mit gleichem Namen zuzüglich aller Signaturen in Spezialisierungen des Datentyps, die den gleichen Namen haben und typverträglich mit einer Signatur in der dynamischen Signatur sind.

Jede dynamische Signatur muss nun ein Gitter bilden, d.h., die für alle Paare (S_i, S_j) von Signaturen aus der dynamischen Signatur muss es eine Signatur S geben, für die gilt

a) S ist typverträglich mit S_i und S_j, und

b) Für jede Signatur S_k, die typverträglich sowohl mit S_i als auch S_j ist, ist auch S mit S_k typverträglich.

Diese Forderung soll anhand von Beispiel 14 illustriert werden.

Beispiel 14. Gegeben seien die Definitionen

```
object type P{
    literals A;
}

object type Q inherits P{
    literals B;
}

object type T{
    struct
        x Integer;
    methods
        virtual M(virtual P, virtual Q);
        virtual M(virtual Q, virtual P);
}

dcl q1 Q := B, q2 Q := B, t T := (. 0 .);

task t.M(q1, q2);
```

Die beiden Deklarationen der Methode M bilden eine dynamische Signatur. Der Aufruf der Methode M wäre dynamisch mehrdeutig, da die dynamischen Typen typverträglich mit beiden Signaturen sind. Durch die Forderung, dass die Signaturen ein Gitter bilden müssen, wird jedoch dieses SDL-Fragment ungültig, da die Signatur M diese Forderung verletzt. Die Korrektur der Spezifikation kann beispielsweise darin bestehen, dass eine weitere Methode

```
virtual M(virtual Q, virtual Q);
```

eingeführt wird, die typverträglich mit beiden anderen Methoden ist, und damit für den Aufruf ausgewählt wird.

6.5 Datentypkonstruktoren

Zusätzlich zu den im Paket predefined (siehe 6.13) definierten Datentypen können neue Datentypen durch Konstruktion aus existierenden Datentypen entstehen. Dabei stehen drei Typkonstruktoren (im Folgenden einfach *Konstruktor* genannt) zur Verfügung:

- Die Elemente der Sorte des Datentyps können durch explizite Aufzählung definiert werden. Diese Elemente heißen *Literale*; der Typ ist ein *Literaltyp*.
- Die Sorte ist das kartesische Produkt von Sorten S_1, S_2, ..., S_n. Der so entstehende Typ heißt *Strukturtyp* (**struct**). Seine Elemente sind n-Tupel. Die Projektionen dieser Tupel heißen *Felder*.
- Die Sorte simuliert die Vereinigung von Sorten S_1, S_2, ..., S_n. Der entstehende Typ heißt *Choice-Typ* (**choice**). Seine Elemente sind (n+1)-Tupel bestehend aus dem Selektor, der die aktuelle *Variante* angibt, sowie je einem Element der Sorten S_1, S_2, ..., S_n.

Diese Festlegung von drei Typkonstruktoren ist eine Einschränkung gegenüber SDL-92, wo eine große Vielfalt anders strukturierter Typen definierbar war. Leider hatte der in SDL-92 verfolgte Ansatz den Nachteil, im Allgemeinen nicht berechenbar und nicht effizient implementierbar zu sein [Sch02]. Die in SDL-2000 getroffene Einschränkung orientiert sich zum einen an den Datentypsystemen gebräuchlicher objektorientierter Sprachen (beispielsweise von C++), andererseits an den Bedürfnissen von ASN.1 [X.680] und seiner Integration in SDL [Z.105].

Die Regel <data type constructor> wählt aus diesen drei Möglichkeiten eine aus:

<data type constructor> = <literal list> | <structure definition> | <choice definition>

Bei Spezialisierung von Datentypen muss der Konstruktor der Spezialisierung der gleiche sein wie der Konstruktor des Basistyps. Die Eigenschaften des Basistyps finden sich dann in der Spezialisierung wieder:

- Die Spezialisierung eines Literaltyps enthält die gleichen Literale wie der Basistyp und kann weitere Literale hinzufügen.
- Die Spezialisierung eines Strukturtyps enthält die gleichen Felder wie der Basistyp und kann weitere Felder hinzufügen.
- Die Spezialisierung eines Choice-Typs enthält die gleichen Varianten wie der Basistyp und kann weitere Varianten hinzufügen.

Literaltypen werden durch die Aufzählung aller Elemente festgelegt. Diese Aufzählung erfolgt durch die Regeln <literal list> und <literal signature>:

<literal list> :: [<visibility>] <literal signature>+

<literal signature> :: {<literal name> | <name class literal> | <named number>}

Als <literal name> können einerseits Bezeichner verwendet werden (einschließlich von Bezeichnern, die nur aus Ziffern bestehen), andererseits aber auch Zeichenketten.

<named number> :: <literal name> <Natural<simple expression>

Aus der Abbildung von ASN.1 auf SDL ergibt sich die Forderung, Literale mit numerischen Werten versehen zu können. Wird lediglich das Konstrukt <literal name> verwendet, so erfolgt die Nummerierung von links beginnend mit 1. Ist eine andere Nummerierung erwünscht, so kann durch die Regel <named number> für ein Literal ein numerischer Wert festgelegt werden. Diese Werte werden dann bei der Nummerierung der anderen Literal ausgelassen.

Mit Hilfe dieser beiden Alternativen von <literal signature> können nur endliche Mengen definiert werden. Zur Konstruktion unendlicher Mengen von Literalen gibt es die Alternative <name class literal>, auf die in Abschnitt 6.7 eingegangen wird.

Die Deklaration eines Typs S als Literaltyp impliziert eine Reihe von Operatoren:

```
"<"( this S, this S )-> Boolean;
">"( this S, this S )-> Boolean;
"<="( this S, this S )-> Boolean;
```

```
">="( this S, this S )-> Boolean;
first          -> this S;
last           -> this S;
succ( this S )-> this S;
pred( this S )-> this S;
num( this S )-> Natural;
```

Die Operatoren <, >, <= und >= legen eine Ordnung auf den Literalen des Typs fest, diese Ordnung entsteht durch die Deklarationsreihenfolge (also unabhängig von den numerischen Werten). Elemente einer Spezialisierung folgen hierbei auf die Elemente des Basistyps. Die Operatoren first und last ermitteln das erste und letzte Element der Sorte in Bezug auf diese Ordnungsrelation. Die Operatoren succ und pred ermitteln das in Bezug auf diese Ordnung nächste beziehungsweise vorangehende Element, dabei gilt das kleinste als sein eigener Vorgänger und das größte als sein eigener Nachfolger. Der Operator num ermittelt die mit dem Literal assoziierte Zahl.

Beispiel 15. Gegeben sei der Literaltyp

```
value type Color{
    literals rot, gruen = 1, blau;
}
```

Für diesen Typ sind die folgenden Ausdrücke wahr:

```
rot < gruen
gruen < blau
num(rot) = 2
num(gruen) = 1
num(blau) = 3
```

Die Definition eines Strukturtyps erfolgt mit Hilfe der Regel <structure definition>. Die Bedeutung von <visibility> wird in Abschnitt 6.8 erläutert

```
<structure definition> :: [<visibility>] <field>*
```

Eine Struktur wird definiert durch die Folge aller Felder der Struktur. Ein Feld ist entweder optional oder obligatorisch:

```
<field> = <optional field> | <mandatory field>
```

Optionale Felder werden in der konkreten Syntax vom Schlüsselwort **optional** eingeleitet, gefolgt von Feldnamen einer Sorte.

```
<optional field> :: <fields of sort>
```

Obligatorische Felder werden ohne spezielles Schlüsselwort definiert. Sie können mit einem Standardwert definiert werden, den das Feld annimmt, wenn bei Konstruktion eines Wertes der Struktur kein Wert angegeben wurde.

```
<mandatory field> :: <fields of sort> [<field default initialization>]

<field default initialization> :: <constant expression>
```

Die Felder einer Struktur werden durch die Regel <fields of sort> definiert. Dabei können in einer derartigen Definition mehrere Felder mit gleicher Sorte und gleicher Sichtbarkeit definiert werden.

```
<fields of sort> ::[<visibility>] <field<name>+ <sort>
```

Beispiel 16. Die folgende Definition des Typs Struktur nutzt diese Syntax vollständig aus:

```
value type Struktur{
    public struct
```

```
        optional private x,y Integer;
        z Integer default 100;
}
```

Dieser Typ Struktur verfügt über drei Felder: x, y und z. Die ersten beiden Felder sind optional und privat. Das letzte ist öffentlich, obligatorisch, und hat standardmäßig den Wert 100.

Eine Strukturdefinition führt implizit zur Erzeugung einer Reihe von Operationssignaturen. Zum einen wird ein Operator Make eingeführt, der einen Wert aus der Angabe aller Felder erzeugt. Für das Beispiel 16 hat dieser Operator die Signatur

```
operators
    public Make(Integer,Integer,Integer) -> Struktur;
```

Beim Aufruf dieses Operators können Argumente weggelassen werden. Die entsprechenden Felder erhalten dann ihren Standardwert. Wenn kein Standardwert definiert war, sind diese Felder undefiniert.

Zum anderen wird für jedes Feld eine lesende und eine schreibende Zugriffsmethode definiert, und, wenn das Feld optional ist, eine Testmethode, die ermittelt, ob das Feld definiert ist. Für das Beispiel 16 haben diese Methoden die Signatur

```
methods
    public virtual xExtract() -> Integer; raises UndefinedField;
    public virtual xModify(Integer);
    public virtual xPresent() -> Boolean;

    public virtual yExtract() -> Integer; raises UndefinedField;
    public virtual yModify(Integer);
    public virtual yPresent() -> Boolean;

    public virtualzExtract() -> Integer;
    public virtual zModify(Integer);
```

Die Methoden, deren Namen auf „Extract“ enden, ermitteln den aktuellen Wert des Felds. Falls der undefiniert ist, lösen sie die vordefinierte Ausnahme UndefinedField aus. Die auf „Modify“ endenden Methoden verändern den Wert. Die auf „Present“ endenden Methoden liefern false, wenn der Wert des Feldes undefiniert ist. Die Methodennamen „Extract“ und „Modify“ werden in der Praxis nie verwendet, da folgende Kurznotationen in Ausdrücken verwendet werden können:

```
dcl var Struktur, i Integer;
task var := (. 3, , 4 .);   /* Kurznotation für Make(3, , 4) */
task var.y := 5;            /* Kurznotation für x.yModify(5) */
task i := var.x;            /* Kurznotation für i := var.xExtract() */
```

Zur Abbildung des ASN.1-Konstrukts **CHOICE** auf SDL wurde in SDL-2000 der Typkonstruktor **choice** eingeführt. Ursprünglich war geplant, die Sorte dieses Typs als eine Vereinigung der Sorten aller Varianten des Choice-Typs zu definieren. Von dieser Strategie wurde Abstand genommen, weil auffiel, dass gewisse Spezialfälle in der englischen Formulierung der Semantik nicht erklärt waren, und nach Korrektur eines derartigen Problems immer wieder neue Probleme mit dieser informalen Definition auftraten. Daraufhin wurde die semantische Fundierung von Choice-Typen revidiert, und sie als eine Kurznotation für einen Strukturtyp eingeführt.

Ein Choice-Typ wird mit Hilfe der Regel <choice definition> eingeführt:

```
<choice definition> ::[<visibility>] <choice of sort>*
```

Die Definition von Choice-Feldern erfolgt ähnlich der von Strukturfeldern, nur dass keine Standardinitialisierungen erlaubt sind und es keine optionalen Felder gibt.

```
<choice of sort> :: [<visibility>] <field<name> <sort>
```

Beispiel 17. Aus der OMG-IDL-Definition für den Typ GIOP::TargetAddress [OMG02a] ergibt sich durch Anwendung des IDL-Mapping [Z.130] der Choice-Typ

```
value type TargetAddress{
    choice
        object_key OctetString;
        profile TaggedProfile;
        ior IORAddressingInfo;
}
```

Für Choice-Typen ist ein Transformationsmodell definiert, welches den Typ in einen Strukturtyp umwandelt, der festhält, welche Variante in einem Wert die aktive ist und welches der Wert dieser Variante ist. Für den Typ aus Beispiel 17 ergeben sich nach Transformation folgende Typen:

```
value type anon{
    struct
        object_key OctetString optional;
        profile TaggedProfile optional;
        ior IORAddressingInfo optional;
}

value type anonPresent{
    literals object_key, profile, ior;
}

value type anonChoice{
    struct
        protected Present anonPresent;
        protected Choice anon;
}

value type TargetAddress inherits anonChoice(
        anonMake = Make,
        anonPresentModify = PresentModify,
        anonPresentExtract = PresentExtract,
        anonChoiceModify = ChoiceModify,
        anonChoiceExtract = ChoiceExtract) {

    operators

        /* Konstruktoren: sie erwarten einen Feldwert und konstruieren den Choice-Wert. */
        object_key(OctetString) -> TargetAddress;
        profile(TaggedProfile) -> TargetAddress;
        ior(IORAddressingInfo) -> TargetAddress;

        /* Ermittlung der gültigen Variante */
        PresentExtract(TargetAddress) -> anonPresent;

    methods
        /* Extraktoren, Modifikatoren, Presence-Test. */
        virtual object_keyExtract() -> OctetString;
        virtual object_keyModify(OctetString);
        virtual object_keyPresent() -> Boolean;

        virtual profileExtract() -> TaggedProfile;
        virtual profileModify(TaggedProfile);
        virtual profilePresent() -> Boolean;

        virtual iorExtract() -> IORAddressingInfo;
        virtual iorModify(IORAddressingInfo);
        virtual iorPresent() -> Boolean;

    /* Die eigentlichen Methodendefinitionen werden hier nur exemplarisch
```

```
        für das Feld profile vorgestellt */
    operator profile(TaggedProfile val) -> TargetAddress{
        return anonMake(profile, Make(,val,));
    }
    operator PresentExtract(TargetAddress t) -> anonPresent{
        return anonPresentExtract(t);
    }
    method virtual profileExtract() -> TaggedProfile{
        return this.anon.profile;
    }
    method profileModify(TaggedProfile val){
        this.present := profile;
        this.anon := Make(, val, );
    }
    method profilePresent()->Boolean{
        return this.anon.profilePresent;
    }
}
```

6.6 Verhalten von Operationen

Sofern es sich nicht um Operationen vordefinierter Typen handelt (siehe Abschnitt 8.4), wird das Verhalten einer Operation durch Operationsdefinitionen festgelegt:

```
<operation definitions> =
    <operation definition> | <external operation definition>
```

Erfolgt die Operationsdefinition nicht im Körper der Datentypdefinition, muss statt der Operationsdefinition ein Knoten <textual operation reference> angegeben werden. Die Parameterliste der Operationssignatur kann weggelassen werden, wenn sich aus dem Namen eindeutig ergibt, welche Operation referenziert wird.

```
<textual operation reference> :: <operation kind> <operation signature>
```

Soll für eine Operation überhaupt kein Verhalten (in SDL) spezifiziert werden, so kann die Operationsdefinition in der Form <external operation definition> angegeben werden.

```
<external operation definition> :: <operation kind> <operation signature>
```

Eine solche Definition verwendet in der konkreten Syntax das Schlüsselwort **external**, wie beispielsweise in der Definition

```
operator fork -> Integer; external;
```

Eine solche externe Operationsdefinition hat keine formale Semantik. Damit hat auch die gesamte SDL-Spezifikation, in der die Operation definiert wird, keine formale Semantik. Mit diesem Konstrukt können beispielsweise Operationen deklariert werden, deren Implementierung in einer anderen Programmiersprache vorliegt.

Alle anderen Operationsdefinitionen folgen der Regel <operation definition>

```
<operation definition> ::
    <package use clause>* <operation heading> <entity in operation>*
    { <operation body> | <statement list> }
```

Mit dem Knoten <package use clause> kann wiederum auf Pakete Bezug genommen werden, die innerhalb der Operationsdefinition verwendet werden. Um welche Operation es sich bei der Operationsdefinition handelt, wird durch den Knoten <operation heading> angegeben.

<operation heading> ::
<operation kind> <operation preamble> <qualifier> <operation name>
<formal operation parameter>* [<operation result>] <raises>

<operation kind> = **operator** | **method**

Zunächst legt der Autor fest, ob es sich um eine Operation oder eine Methode handelt. Innerhalb von Methoden kann auf einen zusätzlichen impliziten Parameter durch das Schlüsselwort **this** zugegriffen werden. Dieser Parameter bezeichnet das Objekt, für das die Methode aufgerufen wurde.

Durch den Knoten <qualifier> wird auf den Datentyp Bezug genommen, dessen Operation definiert wird. Dies ist lediglich bei refererenzierten Operation erforderlich. Auf den Operationsnamen folgen die formalen Operationsparameter:

<formal operation parameter> ::
[<argument virtuality>] <parameter kind> <parameters of sort>

Wie schon in der Definition der Operationssignatur wird durch <argument virtuality> festgelegt, ob ein Argument zur dynamischen Bindung verwendet wird. <parameter kind> legt fest, ob es ein Eingabe-, Ausgabe oder Ein-und-Ausgabeparameter ist. Anders als in der Signatur tragen die Parameter in der Definition jedoch Namen. Der Name des Rückgabewerts ist optional.

<parameters of sort> ::<variable<name>+ <sort>

<operation result> ::[<variable<name>] <sort>

Der Knoten <operation result> bestimmt den Ergebnistyp und gibt optional den Namen der Rückgabevariablen an. Durch Definition einer Rückgabevariablen kann auf die Angabe eines Werts in der **return**-Aktion verzichtet werden; in diesem Fall ist der Wert der Rückgabevariablen das Operationsergebnis.

<raises> deklariert, genau wie in der Signatur, die Menge der Ausnahmen, die diese Operation auslösen kann.

Beispiel 18. Unter Ausnutzung aller optionalen syntaktischen Konstrukte kann eine Operationsdefinition mit folgendem Kopf eingeleitet werden:

```
method virtual protected append(virtual inout Charstring data)
  -> result collector raises OutOfMemory
```

Auf die Knoten <operation heading> folgt eine Liste von Definitionen, die in der Operationsdefinition enthalten und damit nur in der Operation sichtbar sind. Erlaubt sind an dieser Stelle alle Definitionsarten (*entity kinds*), die im Operationskörper verwendet werden können, also im Wesentlichen weitere Datendefinitionen, Variablen und Ausnahmen.

<entity in operation> =
<data definition>
| <variable definition>
| <exception definition>
| <select definition>

Eine Sonderstellung nimmt hier der Knotentyp <select definition> ein, da dieser potentiell beliebige weitere Definitionen enthält. Durch den Knoten <select definition> können, abhängig von konstanten Bedingungen, Alternativen ausgewählt werden (die Kindknoten von <select definition> sind). Im Rahmen einer Transformationsregel wird die Bedingung in der <select definition> ausgewertet und dann dieser Knoten entweder durch eine leere Liste oder durch die enthaltenen Definitionen.

Der eigentliche Körper der Operation wird durch den Knoten <operation body> definiert:

<operation body> ::

[<on exception>] <start> {<free action> | <exception handler>}*

Dabei gibt der optionale Knoten <on exception> an, welche Ausnahmebehandlung für den gesamten Graph des Operators gelten soll. Der Knoten <start> gibt die Starttransition an. Anders als bei Zustandsautomaten (Agenten und Zustandstypen) üblich besteht der Graph eines Operators lediglich aus der Starttransition. Der Operatoralgorithmus kann also nicht in einem Zustand verharren.

Allerdings können durch **join**-Konstrukte Marken angesprungen werden, die innerhalb von Knoten des Typs <free action> definiert sind. Außerdem kann durch <on exception>-Knoten auf Ausnahme-Behandler Bezug genommen werden, die in den Knoten <exception handler> definiert werden.

Die Definition der Grammatikregeln für <start>, <free action> und <exception handler> wird hier nicht angegeben. Die Ausdruckskraft ist die gleiche, wie sie auch in der Definition von Agenten verwendet werden kann, mit folgenden Einschränkungen:

- Es dürfen keine Zustände definiert werden.
- Es dürfen keine Knoten <imperative expression> (also **now**, Variablenimport, **self**, **offspring**, **sender**, **active**, **any**, und **state**) verwendet werden.
- Es dürfen nur Bezeichner verwendet werden, die innerhalb des Operators definiert wurden, sowie Bezeichner für Synonyme, Literale, Operationen, Ausnahmen und Sorten.

Beispiel 19. Der oben angegebene Operator append könnte wie folgt definiert sein:

```
object type collector
{
    struct
        contents Charstring;
    method virtual protected append(in virtual Charsrtring data)
        ->result collector raises OutOfMemory
    {
        if (length(data) > 1000) raise OutOfMemory;
        result := this;
        contents := contents // data;
        return;
    }
}
```

Trotz der Einschränkung der möglichen Aktionen innerhalb einer Operationsdefinition können Operationsdefinitionen nun (anders als in SDL-92) Seiteneffekte haben. So ändert die Methode append aus Beispiel 19 den Zustand des Objekts, an dem sie gerufen wird. Dabei werden unter Umständen auch Variablen des rufenden Prozesses geändert.

Beispiel 20. Zur Definition von Transitionen können in SDL sogenannte *continuous signals* verwendet werden. Das sind Ausdrücke, die in einem Zustand immer wieder bewertet werden, bis sie erfüllt sind – dann wird die mit dem Signal assoziierte Transition ausgeführt. Unter Verwendung der Methode append aus Beispiel 19 kann man folgende Transition definieren:

```
dcl the_collector collector;
/* ... */

state EinZustand;
    provided length(the_collector.append('Mehr Text').data) > 20;
        stop;
```

Hier wird im Zustand EinZustand verharrt, bis die Länge der Zeichenkette the_collector.data größer als 20 wird. In SDL-92 hätte dieser Ausdruck, ist er initial falsch, seinen Wert nur dadurch ändern können, dass eine externe Änderung der Variablen erfolgt. Mit SDL-92 hat die

Berechnung einen Seiteneffekt, und wiederholte Berechnung führt dazu, dass der Ausdruck schließlich wahr wird. Für *continuous signals* stellt sich nun die Frage, wie oft der Ausdruck berechnet werden soll, während der Prozess im Zustand EinZustand verharrt.
In Analyse dieser Frage wurde im Standardisierungsgremium zunächst diskutiert, Methoden in solchen Zusammenhängen zu verbieten. Allerdings hätte man dann auch Operatoren verbieten müssen, die Methoden rufen. Da auch ein einfacher lesender Feldzugriff ein Methodenruf ist, wäre die Sprache durch diese Forderung auf unakzeptable Weise eingeschränkt worden.
Alternativ zu dieser Strategie wurde vorgeschlagen, dass Programmen eine statische Bedingung auferlegt wird, in continuous signals keine Zustandsänderungen vorzunehmen. Die Bestimmung, ob ein System diese Bedingung erfüllt, kommt allerdings im Allgemeinen dem Halteproblem gleich, weshalb von dieser Forderung wieder Abstand genommen wurde.
Schließlich wird nun in einer nicht-normativen Bemerkung im Text des SDL-Standards auf dieses Problem hingewiesen und erklärt, dass ein Programm, dass in den Ausdrücken von continuous signals Seiteneffekte enthält, undefiniertes Verhalten hat. Die formale Semantikdefinition beschreibt das genauer: Die Berechnung des Ausdrucks kann beliebig oft erfolgen, bis er erfüllt ist.

6.7 Definition von Literalmengen durch Muster

Zur Definition von Sorten mit (potentiell unendlich) vielen Literalen gibt es in SDL-92 sogenannte Namensklassen (*name classes*). Durch eine spezielle Syntax kann man damit eine Menge von Literalen beschreiben, die alle einem regulären Ausdruck [ASU86] entsprechen und damit zusammen eine reguläre Sprache bilden.

Beispiel 21. Mit dieser Technik werden die Literale der Sorte Integer definiert

```
value type Integer;
  literals unordered nameclass (('0':'9')*) ('0':'9'));
  ...
endvalue type;
```

Die Regel legt fest, dass zur Sorte Integer Literale gehören, die aus beliebig vielen Ziffern bestehen, gefolgt von einer einzelnen Ziffer. Das Zeichen '*' bedeutet dabei den Kleene-Stern. Das Zeichen ':' kennzeichnet eine Menge von Zeichen eines Intervalls, entsprechend der Ordnung der Zeichen innerhalb der Sorte Character. Das Schlüsselwort **unordered** darf nur im Annex D verwendet werden, der die vordefinierten Datentypen erklärt.

Während der Entwicklung von SDL-2000 zeigte sich, dass dieses Konstrukt aus SDL-92 zwar in die neue Sprachversion übernommen werden sollte, aber dort nicht die gleiche Ausdruckskraft haben kann, wie das SDL-92-Konstrukt. So muss beispielsweise definiert werden, wie sich die Literale in Bezug auf die Operatoren verhalten. In SDL-92 wurden für die Daten des Typs Integer zu diesem Zweck folgende Axiome formuliert:

```
/* definition of literals */
    <<type Integer>> 2== 1 + 1;
    <<type Integer>> 3== 2 + 1;
    <<type Integer>> 4== 3 + 1;
    <<type Integer>> 5== 4 + 1;
    <<type Integer>> 6== 5 + 1;
    <<type Integer>> 7== 6 + 1;
    <<type Integer>> 8== 7 + 1;
    <<type Integer>> 9== 8 + 1;

/* literals other than 0 to 9 */
```

```
for all a,b,c in Integer nameclass (
    spelling(a) == spelling(b) // spelling(c),
    length(spelling(c)) == 1       ==> a == b * (9 + 1) + c;
);
```

Zusammen mit den anderen Axiomen für Addition und Multiplikation definieren diese Axiome das Dezimalsystem: Die Literale 2 bis 9 ergeben sich jeweils durch Addition von 1 aus ihrem Vorgänger. Die Literale aus mehreren Ziffern sind äquivalent zu dem Ausdruck 10*b+c, wobei c die letzte Ziffer des Literals ist. Der Operator spelling erlaubt die Umrechnung eines Literals in seine Zeichenfolge.

In SDL-2000 müssen Operatoren eine algorithmische Definition erhalten, das heißt, ein Algorithmus für die Addition müsste den Nachfolger des Literals bestimmen, welches als Argument übergeben wurde. Es gibt jedoch keine inverse Operation zum Operator spelling.

Um die Namensklassen praktisch einsetzbar zu machen, wurde das neue Konzept der Namensklassenoperationen definiert, bei dem durch eine einzige Definition eine große Zahl von Operationen definiert werden. Dieses Konzept ist in der abstrakten Syntax 0 durch den Knoten <name class operation> definiert. Die aus SDL-92 übernommenen Namensklassenliterale werden durch Knoten des Typs <name class literal> definiert:

```
<name class operation> :: <name> <regular expression>

<name class literal> ::<regular expression>
```

In beiden Konstrukten wird ein regulärer Ausdruck angegeben. Für <name class operation>-Knoten gibt es zusätzlich noch einen „internen“ Namen der Operation. In der konkreten Syntax wird eine solche Operation etwa wie folgt definiert:

```
operator vokal_endend in nameclass ('a':'z')+ -> Boolean {
    dcl sp Charstring := spelling(vokal_endend);
    dcl last Character := sp[length(sp)];
    return last = 'a' or last = 'e' or last = 'i' or last = 'o' or last = 'u';
}
```

Mit diesem Namensklassenoperator werden Operatoren definiert, die sämtlich aus Kleinbuchstaben bestehen. Der Wert des Operators ist „wahr“, wenn der Operator mit einem Vokal endet. Dazu muss in dem Operator seine Schreibweise ermittelt werden. An den Operator **spelling** wird hierzu der symbolische Operationsname übergeben. Diese wird dann durch die Zeichen des tatsächlichen Operatornamens ersetzt.

Namensklassenoperatoren müssen null-stellige Operatoren sein. Sie dürfen also keine Methoden sein und keine Argumente haben.

Die regulären Ausdrücke sowohl für Operationen als auch für Literale sind durch den Knoten <regular expression> repräsentiert:

```
<regular expression> ::<or partial regular expression>+
```

Mit einem regulären Ausdruck wird eine Sprache definiert. Man sagt, dass ein Literal *auf den Ausdruck passt*, wenn es zu dieser Sprache gehört.

Die einzelnen Teilausdrücke werden in der konkreten Syntax durch das Schlüsselwort **or** verknüpft, welches die Vereinigung von zwei Teilsprachen definiert.

Beispiel 22. Die Literale der Sorte Real werden durch die Namensklasse

```
literals unordered nameclass
    ( ('0':'9')* ('0':'9') ) or ( ('0':'9')* '.'('0':'9')+ );
```

definiert. Das heißt, ein Real-Literal besteht entweder nur aus Ziffern, oder aus Ziffern, gefolgt von einem Punkt, gefolgt von wenigstens einer Ziffer.

Die verschiedenen Alternativen im regulären Ausdruck werden durch den Knoten <or partial regular expression> repräsentiert:

```
<or partial regular expression> ::<partial regular expression>+
```

Dieser Knoten wiederum ist eine Liste von <partial regular expression>. Diese Liste bedeutet Hintereinanderschreibung: Ein Literal passt auf den regulären Ausdruck, wenn es sich so in Teilliterale zerlegen lässt, dass jeder Teil auf den entsprechenden Knoten <partial regular expression> passt, und die Verkettung der Teilliterale das Gesamtliteral ergibt. Jeder Teil wird durch die Regel

```
<partial regular expression> ::
        <regular element> [<name> | <plus sign> | <asterisk>]
```

definiert. Ein solcher Teilausdruck ist ein Knoten vom Typ <regular element>, dem optional eine natürliche Zahl (in der abstrakten Syntax der Knoten <name>), ein Pluszeichen oder ein Stern folgt.

Der Stern bedeutet den Kleene-Stern. Auf den regulären Ausdruck passen Literale, die sich durch beliebige (auch nullfache) Wiederholung des regulären Ausdrucks <regular element> ergeben.

Das Pluszeichen bedeutet, dass die Wiederholung wenigstens einfach sein muss. Die Zahl bedeutet, dass die Wiederholung genau so oft erfolgen muss, wie die Zahl es angibt.

Beispiel 23. Mit folgendem Konstrukt werden Literale definiert, die aus einer geraden Anzahl von Zeichen bestehen:

```
literals nameclass (('a':'z')2)+
```

Die Grundelemente eines regulären Ausdrucks sind die Knoten <regular element>:

```
<regular element> =
        <regular expression>
    |   <character string>
    |   <regular interval>
```

Als erste Alternative kann ein beliebiger regulärer Ausdruck Grundelement sein. In der konkreten Syntax wird das durch Klammern notiert. Durch diese Konstruktion können verschachtelte reguläre Ausdrücke definiert werden. Die Alternative <character string> erlaubt die Verwendung eines einzelnen Zeichens oder einer Zeichenkette als Element eines regulären Ausdrucks. Der Knoten <regular interval> erlaubt die Angabe eines Intervalls von Zeichen:

```
<regular interval> ::<character string> <character string>
```

In der konkreten Syntax werden die beiden Zeichenketten durch das Zeichen ':' getrennt. Beide Zeichenketten müssen die Länge 1 haben. Das Intervall bezeichnet dann die Menge aller Zeichen zwischen dem ersten und zweiten Kindknoten des Intervalls, einschließlich der Intervallenden.

Nur innerhalb eines Namensklassenoperators kann der Knoten <spelling term> verwendet werden, der in der konkreten Syntax durch den Operator spelling notiert wird.

```
<spelling term> :: <name>
```

Der in diesem Knoten angegebene Name muss die Operation bezeichnen; die Angabe eines Namens ist also, genau genommen, unnötig.

6.8 Sichtbarkeit

Durch die Deklaration von Sichtbarkeit kann der Zugriff auf Operatoren und Literale eingeschränkt werden. Für Felder von Struktur- oder Choice-Typen wird die Sichtbarkeit des Felds auf die der impliziten Operationen übertragen.

Der Knoten <visibility> kann drei mögliche Werte annehmen:

<visibility> = **public** | **protected** | **private**

Hat ein Operator (oder ein Literal) die Sichtbarkeit **private**, so kann er nur innerhalb des Datentyps verwendet werden. Außerhalb des Datentyps ist er nicht sichtbar und in Spezialisierungen wird er in einen anonymen Operator umbenannt. Hat ein Operator die Sichtbarkeit **protected**, so ist er nur innerhalb des Datentyps sichtbar. Da Ableitungen eine Kopie des Operators erhalten, ist in diesen Ableitungen die Kopie sichtbar. Operatoren mit der Sichtbarkeit **public** unterliegen keinen Einschränkungen.

Das Konzept der Sichtbarkeit in SDL weicht vom Konzept der Zugriffskontrolle, etwa in C++, ab: Während in C++ eine Methode, die als **protected** deklariert ist, zwar durchaus im Prozess der Namensauflösung noch gefunden werden kann, ist es ein Fehler, wenn diese Methode verwendet wird. In SDL stehen die nicht sichtbaren Operatoren bei der Namensauflösung gar nicht erst zur Verfügung.

6.9 Syntype-Sorten

Zur Definition von Teilbereichs- und Aliastypen gibt es in SDL das Konstrukt <syntype>:

<syntype> =<identifier>

Ein Knoten des Typs <syntype> ist ein Bezeichner, der sich auch einen Knoten des Typs <syntype definition> bezieht:

<syntype definition> ::

<package use clause>* { <syntype definition gen syntype> | <syntype definition gen type preamble> }

Kern einer solchen Definition ist ein Knoten <constraint>, der die Teilbereichseinschränkung definiert. Um eine solche Einschränkung auch in Objekt- und Wertetypdefinitionen vornehmen zu können, sind in SDL-2000 zwei Formen von Syntype-Definitionen vorgesehen. Die erste Form wird durch den Knoten <syntype definition gen syntype> definiert:

<syntype definition gen syntype> ::

<syntype<name> <parent sort identifier>

[<default initialization>] [<constraint>]

<parent sort identifier> =<sort>

In der konkreten Syntax wird diese Form durch das Schlüsselwort **syntype** eingeleitet. Der Knoten <parent sort identifier> gibt die einzuschränkende Basissorte an, der Knoten <default initialization> einen optionalen Standardwert für Variablen dieser Sorte, und der Knoten <constraint> eine optionale Teilbereichseinschränkung (ohne diese Einschränkung wird ein Aliastyp definiert).

Beispiel 24. Der Typ *Natural* wird durch die Definition

syntype Natural = Integer **constants** >= 0; **endsyntype** Natural;

beschrieben. Die Basissorte ist die Sorte Integer; die Einschränkung schränkt den Typ auf die nichtnegativen Zahlen ein. Ein Standardwert ist nicht definiert.

Alternativ kann durch den Knoten <syntype definition gen type preamble> eine Einschränkung direkt in einem Objekt- oder Wertetyp vorgenommen werden. Die syntaktische Struktur gleicht dem Knoten <data type definition>, hier ist jedoch als zusätzlicher Kindknoten <constraint> erforderlich.

<syntype definition gen type preamble> ::

<type preamble> <data type heading> [<data type specialization>]

[<data type definition body>] <constraint>

In der konkreten Syntax wird dieses Konstrukt von <data type definition> dadurch unterschieden, dass in dem einen Konstrukt <constraint> vorhanden sein muss und es in dem anderen Konstrukt nicht vorhanden sein darf. Bei <syntype definition gen type preamble> handelt es sich um eine Kurznotation für einen anonymen Datentyp und eine Syntypedefinition, die diesen anonymen Typ einschränkt.

Die eigentliche Einschränkung des Typs erfolgt durch den Knoten <constraint>:

<constraint> = <range condition> | <size constraint>

Dieser Knoten kann zwei Ausprägungen haben: Einerseits kann durch den Knoten <range condition> eine Einschränkung der möglichen Werte auf eine Menge von Intervallen ausgedrückt werden:

<range condition> ::<range>+

In der konkreten Syntax werden die Intervalle durch das Schlüsselwort **constants** eingeleitet und durch Kommata getrennt. Jedes dieser Intervalle ist durch einen Knoten <range> definiert:

<range> =<closed range> | <open range>

Ein Knoten <range> definiert ein Intervall, dieses kann entweder ein halb-offenes Intervall (<open range>) oder ein geschlossenes Intervall (<closed range>) sein. Der Fall eines einzelnen Elements wird hier syntaktisch ebenfalls dem offenen Intervall zugeordnet:

<open range> =<constant> | <open range gen greater than or equals sign>

Falls das Intervall nicht aus nur einer Konstanten besteht, wird es durch Angabe eines Relationsoperators und einer Konstanten definiert:

<open range gen greater than or equals sign> ::

{ <equals sign> | <not equals sign> | <less than sign> | <greater than sign> |

<less than or equals sign> | <greater than or equals sign> } <constant>

Dabei sind durch das Intervall all die Elemente im Teilbereichstyp enthalten, die entsprechend dem Relationsoperator gleich („=“), ungleich („/=“), kleiner („<„), größer („>“), kleiner-gleich („<=“) oder größer-gleich („>=“) der Konstante sind. Der Fall, dass kein Operator angegeben ist, entspricht damit semantisch dem Fall, dass der Relationsoperator <equals sign> ist.

Der Knoten <closed range> repräsentiert ein geschlossenes Intervall, bei dem also alle Elemente zum Intervall gehören, die größer-gleich einer unteren Grenze und kleiner-gleich einer oberen Grenze sind. Dieser Knoten enthält in der abstrakten Syntax 0 als Kindknoten die beiden Intervallgrenzen:

<closed range> ::<constant> <constant>

In der konkreten Syntax können sie wahlweise mittels „:“ oder „..“ getrennt sein.

Beispiel 25. Ein Index-Typ, der auf die Werte 1 bis N beschränkt sein soll, kann durch

```
syntype Index = Integer constants 1:N; endsyntype;
```

definiert werden.

Neben der Einschränkung einer Syntype-Sorte auf eine Vereinigung von Teilintervallen kann die Sorte auch noch durch eine Längenbeschränkung definiert werden. Das setzt natürlich voraus, dass für die Sorte der Begriff der Länge, also ein Operator Length definiert ist. Dieses Konstrukt wurde in SDL-2000 aufgenommen, um das entsprechende Konstrukt von ASN.1 abbilden zu können.

In der konkreten Syntax wird eine Längenbeschränkung durch das Schlüsselwort **size** eingeleitet. In der abstrakten Syntax wird sie durch den Knoten <size constraint> repräsentiert:

```
<size constraint> :: <range condition>
```

Die eigentliche Beschränkung muss dann einen Teilbereich von Integer definieren. Sie wird in der konkreten Syntax in Klammern geschrieben.

Beispiel 26. Der in [Q.1248] definierte ASN.1-Typ

```
AlertingPattern ::= OCTET STRING(SIZE (3))
```

kann in SDL-2000 durch den Typ

```
syntype AlertingPatter = OctetString size (<=3); endsyntype;
```

ausgedrückt werden.

Zwischen einer Syntype-Sorte und ihrer Basissorte besteht Zuweisungskompatibilität: Die Sorte ist tatsächlich dieselbe. Die dynamische Semantik von Syntypes ergibt sich aus zusätzlichen Laufzeitüberprüfungen, die für Variablen von Syntype-Sorten durchgeführt werden. Immer, wenn ein Wert der Basissorte eine Variable eines Syntypes (oder einen Operator- oder Signalparameter usw.) durchgeführt wird, erfolgt eine Bereichsüberprüfung. Liegt der Wert dann außerhalb des Bereichs, wird die vordefinierte Ausnahme OutOfRange ausgelöst.

6.10 Synonyme

Synonyme sind symbolische Namen für Konstanten. Sie werden mit Hilfe des Knotens <synonym definition> definiert:

```
<synonym definition> :: <synonym definition item>+
```

In der konkreten Syntax wird die Definition durch das Schlüsselwort **synonym** eingeleitet, und durch Kommata getrennt. Es gibt zwei Formen von Synonymdefinitionen, interne und externe:

```
<synonym definition item> =
      <internal synonym definition item> | <external synonym definition item>
```

Eine internes Synonym wird durch den Knoten <internal synonym definition> definiert:

```
<internal synonym definition item> ::
   <name> [<sort>] <constant expression>
```

In dieser Grammatikregel gibt <name> den Namen des Synonyms an und <constant expression> seinen Wert. Ist aus dem Ausdruck, der den Wert bestimmt, die Sorte des Werts nicht eindeutig ermittelbar (etwa weil mehrdeutige Literale verwendet werden), so muss die Sorte angegeben, ansonsten kann sie weggelassen werden. In der konkreten Syntax werden Name und Wert durch ein Gleichheitszeichen getrennt (siehe Beispiel 27).

Eine externe Synonymdefinition wird durch den Knoten <external synonym definition> definiert:

<external synonym definition item> ::<name> <sort>

In dieser Regel sind nur der Name und die Sorte des Synonyms angegeben. Der Wert des Synonyms ist extern in Bezug auf das SDL-System. Eine Spezifikation, in der dieses Konstrukt verwendet wird, hat keine formale Semantik.

Beispiel 27. Die folgende Definition führt zwei Synonyme ein, eines vom Typ Integer, das andere vom Typ Boolean:

synonym a Integer = 100, b = True;

Die Angabe einer Sorte für b ist nicht erforderlich, da sich das Literal True eindeutig der Sorte Boolean zuordnen lässt[18].

Durch Einführung objektorientierter Typen sowie von Ausnahmen in SDL-2000 ergab sich die Frage, wann genau die Auswertung von Synonymen erfolgt. Die Lösungsstrategie von SDL-92 (Ausdrücke werden nicht berechnet, sondern sind in Äquivalenzklassen enthalten) ließ sich nicht mehr anwenden, da die Berechnung eine Ausnahme auslösen könnte, und deshalb geklärt sein muss, wann diese Ausnahme auftritt.

In Analogie zu anderen Sprachen mit berechneten Konstanten (etwa C++ und Java) wurde deshalb festgelegt, dass die Synonyme (und alle anderen konstanten Ausdrücke) zum Systemstart in unspezifizierter Reihenfolge berechnet werden. Anders als in diesen Sprachen können sich aber nicht alle Agenten globale Variablen teilen, deshalb wurde festgelegt, dass bei jeder Verwendung der Konstanten eine Kopie erzeugt wird. Ist die Sorte des Synonyms also eine Objektsorte, so ergibt sich bei mehrfacher Verwendung des Synonyms jedes Mal ein anderes Objekt.

6.11 Ausdrücke

Durch Ausdrücke wird in Agenten, Operatoren, Prozeduren usw. der Aufruf von Operatoren formuliert. Zusätzlich kann man in Ausdrücken eine Reihe von Spezialfunktionen in einem Agenten benutzen.

Alle Ausdrücke folgen der Produktion <expression>. Dies ist die Vereinigung verschiedener Formen von Ausdrücken:

<expression> =
    <create expression>
  | <value returning procedure call>
  | <range check expression>
  | <binary expression>
  | <equality expression>
  | <expression gen primary>

In der konkreten Syntax wird über Grammatikregeln der Vorrang von Operatoren festgelegt. In der abstrakten Syntax 0 kann hierauf verzichtet werden, indem für binäre und unäre Operatoren Knotentypen definiert werden, die zwei beziehungsweise einen Knoten vom Typ <expression> als Kindknoten besitzen:

<binary expression> ::
    <expression>
    { <implies sign> | **or** | **xor** | **and**

18. Genau genommen hängt das vom Programm ab; ein Programm könnte eigene Literale oder Synonyme True definieren.

| <greater than sign> | <greater than or equals sign> | <less than sign>
| <less than or equals sign> | **in** | <plus sign> | <hyphen>
| <concatenation sign> | <asterisk> | <solidus> | **mod** | **rem** }
<expression>

<expression gen primary> ::[<hyphen> | **not**] <primary>

Jeder dieser Knoten hat zusätzlich einen weiteren Kindknoten. Tabelle 3 gibt die konkrete Syntax sowie die übliche Bedeutung der Operatoren an. Die Semantik eines Operators hängt natürlich von den Parametertypen ab. Die in der Tabelle angegebene Bedeutung ist also lediglich eine Konvention, die für die meisten vordefinierten Datentypen eingehalten wurde.

Tabelle 3: Unäre und binäre Operatoren

Konkrete Syntax	Abstrakte Syntax 0	Bedeutung
=>	<implies sign>	logische Implikation
or	**or**	logisches Oder; Vereinigung
xor	**xor**	logisches Exklusiv-Oder
and	**and**	logisches Und; Durchschnitt
>	<greater than sign>	arithmetisch Größer Als; echte Obermenge
>=	<greater than or equals sign>	arithmetisch Größer Oder Gleich; Obermenge
<	<less than sign>	arithmetisch Kleiner; echte Teilmenge
<=	<less than or equals sign>	arithmetisch Kleiner Oder Gleich; Teilmenge
in	in	Element von
+	<plus sign>	Addition
-	<hyphen>	Subtraktion; arithmetische Negation (unär)
//	<concatenation sign>	Stringverkettung
*	<asterisk>	Multiplikation
/	<solidus>	Division
mod	**mod**	Modulobildung
rem	**rem**	Ganzzahlrest
not	**not**	logische Negation

Die Semantik dieser Operatoren wird durch Transformation auf die Semantik von Operatorrufen zurückgeführt, so entspricht der Ausdruck 1 + 2 dem Operatorruf "+"(1, 2). Diese Transformation wird in Abschnitt 7.3.3 vorgestellt.

6.11.1 Primärausdrücke

Die Operanden eines binären oder unären Operators sind Primärausdrücke, sofern sie nicht wiederum Operatoren enthalten. Primärausdrücke werden durch die Regel <primary> ausgedrückt:

```
<primary> =
        <operation application>
    |   <literal>
    |   <expression>
    |   <conditional expression>
    |   <spelling term>
    |   <extended primary>
    |   <active primary>
    |   <synonym>
```

Von diesen Alternativen wurde <spelling term> im Abschnitt 6.7 erläutert. Durch die Aufnahme von <expression> können auch unäre Operatoren auf beliebige Ausdrücke angewendet werden. In der konkreten Syntax sind hierzu Klammern erforderlich.

In den folgenden Absätzen wird die Bedeutung der übrigen Syntaxregeln erläutert.

Der Knoten <operation application> repräsentiert einen Operator- oder Methodenruf.

```
<operation application> = <operator application> | <method application>
```

Ein Operatorruf wird in der konkreten Syntax durch Operatornamen und (optional) eine kommagetrennte Argumentliste in Klammern notiert. In der abstrakten Syntax 0 wird er durch den Knoten <operator application> repräsentiert:

```
<operator application> :: <operation identifier> [<actual parameter>]*
```

In einem Methodenruf wird zusätzlich in der konkreten Syntax vor dem Ausdruck das Zielobjekt des Methodenrufs angegeben. Dieses wird vom Methodennamen durch einen Punkt oder ein Ausrufezeichen abgetrennt. In der abstrakten Syntax 0 wird ein solcher Ruf durch <method application> repräsentiert:

```
<method application> :: <primary> <operation identifier> [<actual parameter>]*
```

Beispiel 28. Nach Definition des Typs TargetAddress aus Beispiel 17 auf Seite 73 kann man folgende Definitionen und Ausdrücke notieren:

```
dcl ta TargetAddress, os OctetString;

ta.iorPresent  /* Methodenruf ohne Argumente vom Typ Boolean */
ta!object_keyModify(os) /* Methodenruf mit einem Argument vom Typ TargetAddress */
```

Literale sind Primärausdrücke, die Elemente eines Literaltyps bezeichnen. Sie werden durch Angabe des Literalbezeichners notiert:

```
<literal> = <literal identifier>

<literal identifier> =<identifier>
```

Der Knoten <conditional expression> bezeichnet bedingte Ausdrücke:

```
<conditional expression> ::

        <expression> <consequence expression> <alternative expression>

<consequence expression> =<expression>

<alternative expression> =<expression>
```

In der konkreten Syntax werden die Teilausdrücke durch die Schlüsselwortfolge **if**, **then**, **else** und **fi** abgetrennt. Zur Interpretation eines bedingten Ausdrucks wird zunächst der erste Kind-Ausdruck von <conditional expression> interpretiert. Liefert dieses den Wert True (vom Typ

Boolean), so wird der Knoten <consequence expression> interpretiert, liefert dieses den Wert False, so wird der Knoten <alternative expression> interpretiert.[19]

Zur Vereinfachung der Benutzung von Strukturtypen und Feldern bietet SDL Konstrukte, die zusammengefasst als erweiterte Primärausdrücke (<extended primary>) bezeichnet werden:

```
<extended primary> =
        <indexed primary>
    |   <field primary>
    |   <composite primary>
```

Mit dem Knoten <indexed primary> wird ein indizierter Zugriff auf ein Feld ausgedrückt:

```
<indexed primary> :: <primary> [<actual parameter>]+
```

Dabei bezeichnet <primary> das Feld (etwa vom vordefinierten Typ Array). Die Indizes werden in den Knoten <actual parameter> dargestellt. In der konkreten Syntax sind sie kommagetrennt und in eckige oder runde Klammern eingeschlossen.

Dieser Ausdruck ist eine Kurznotation für einen Operatoraufruf des Operators Extract.

Beispiel 29. Der Ausdruck a[3] ist gleichbedeutend mit dem Ausdruck Extract(a, 3).

Zum Zugriff auf Felder eines Strukturtyps dient das Konstrukt <field primary>:

```
<field primary> :: [<primary>] <field name>

<field name> = <name>
```

Der Primärausdruck, der den Strukturwert bezeichnet, kann weggelassen werden, wenn der Strukturzugriff innerhalb einer Methode des Strukturtyps erfolgt. Der Primärausdruck ist dann implizit der Ausdruck **this**. Ein Feldzugriff ist eine Kurznotation für den Aufruf eines Operators (siehe Abschnitt 6.5).

Zur Erzeugung von Strukturwerten dient der Ausdruck <composite primary>:

```
<composite primary> ::<qualifier> [<actual parameter>]+
```

In der konkreten Syntax werden die aktuellen Parameter in das Klammernpaar „(.“ und „.)“ eingefasst. Der Knoten <qualifier> kann auch leer sein. Ein Knoten <composite primary> ist eine Kurznotation für den Aufruf des Operators Make.

Beispiel 30. Gegeben sei der Strukturtyp aus Beispiel 16 auf Seite 71 sowie die Definition

```
dcl s Struktur;
```

Die Zuweisung

```
task s := (. 1, 2, 3 .);
```

ist äquivalent zur Zuweisung

```
task s := <<type Struktur>>Make(1, 2, 3);
```

Schließlich enthält die Regel <primary> die Alternative <active primary>, die Ausdrücke beschreibt, deren Interpretation über die reine Datentypsemantik hinausgeht. Dabei gibt es zwei Formen aktiver Primärausdrücken: Variablenzugriffe und imperative Ausdrücke:

```
<active primary>  =<variable access> | <imperative expression>
```

Der Variablenzugriff wird im Knoten <variable access> repräsentiert. Dabei handelt es sich entweder um einen Variablennamen oder um das Schlüsselwort **this**:

```
<variable access>  ::{ <identifier> | this }
```

19. Terminiert die Berechnung der Bedinung nicht oder liefert sie eine Ausnahme, so wird keiner der anderen Teilausdrücke interpretiert.

Die Bedeutung von **this** als Variablenname ergibt sich durch ein Transformationsmodell: Das Schlüsselwort wird durch den Namen des impliziten ersten Parameters von Methoden ersetzt. Demzufolge ist die Verwendung von **this** nur innerhalb von Methoden zulässig. In der dynamischen Semantik bedeutet ein Variablenzugriff die Ermittlung des letzten Werts der Variablen im gegebenen Kontext (Agent, Operation, usw.). Hatte die Variable noch nie einen Wert, ergibt der Zugriff die Ausnahme UndefinedVariable.

Die letzte Alternative der Regel <primary> ist der Knoten <synonym>, der den Zugriff auf ein Synonym bedeutet:

<synonym> ::<identifier>

Zur Interpretation dieses Knotens wird der während der Systemerzeugung initialisierte Wert des Synonyms kopiert; Ergebnis des Ausdrucks ist diese Kopie.

6.11.2 Imperative Ausdrücke

Imperative Ausdrücke nehmen, wie Variablenzugriffe, ebenfalls Bezug auf den Systemzustand. Sie sind durch die Regel <imperative expression> definiert:

<imperative expression> =
    <now expression>
  | <import expression>
  | <pid expression>
  | <timer active expression>
  | <any expression>
  | <state expression>

Der Knoten <now expression> wird in der konkreten Syntax durch das Schlüsselwort now repräsentiert, die Interpretation dieses Ausdrucks liefert die aktuelle Systemzeit (vom vordefinierten Typ Time):

<now expression> ::

Der Knoten <import expression> repräsentiert den Zugriff auf eine entfernte Variable:

<import expression> :: <identifier> <communication constraint>*

Der Knoten <identifier> bezeichnet dabei den Namen der entfernten Variablen. Die Liste von <communication constraint> erlaubt die Angabe des entfernten Prozesses per Name, Pid-Ausdruck oder Einschränkung der möglichen Kommunikationswege mit dem entfernten Agenten.

Die Semantik von <import expression> wird vollständig durch ein Transformationsmodell definiert, bei dem der importierende Agent ein Signal an den exportierenden Prozess sendet und dieser den Wert der exportierten Variable mit einem weiteren Signal an den Importeur zurückgibt.

Die Regel <pid expression> ist die Vereinigung von mehreren Ausdrücken, die alle vom vordefinierten Typ Pid sind:

<pid expression> =
    <self expression>
  | <parent expression>
  | <offspring expression>
  | <sender expression>

In der konkreten Syntax werden diese Ausdrücke durch die Schlüsselwörter **self**, **parent**, **offspring** und **sender** notiert. In der abstrakten Syntax 0 werden sie durch parameterlose Knoten repräsentiert:

<self expression> ::

<parent expression> ::

<offspring expression> ::

<sender expression> ::

Die dynamische Semantik dieser Ausdrücke besteht in der Ermittlung einer Agentenidentifikation, und zwar der des aktuellen Agenten (<self expression>), des Agenten, der den aktuellen Agenten erzeugt hat (<parent expression>), des Agenten, der zuletzt vom aktuellen Agenten erzeugt wurde (<offspring expression>) und des Agenten, von dem zuletzt ein Signal konsumiert wurde (<sender expression>).

Der Ausdruck <timer active expression> ermittelt, ob ein als Argument angegebener Zeitgeber aktiv ist. Besitzt der Zeitgeber Argumente, so müssen auch diese angegeben werden. In der konkreten Syntax wird der Bezeichner des Zeitgebers in Klammern hinter das Schlüsselwort **active** geschrieben. Die Argumente folgen kommagetrennt in Klammern hinter dem Bezeichner.

<timer active expression> ::<identifier> <expression>*

Der Knoten <any expression> repräsentiert die nicht-deterministische Auswahl eines Werts aus einer Sorte. Handelt es sich bei der Sorte um eine Syntype-Sorte, wird ein Wert aus der eingeschränkten Sorte gewählt. In der konkreten Syntax wird die Sorte in Klammern hinter das Schlüsselwort **any** geschrieben.

<any expression> ::<sort>

Letzte Alternative der Regel <imperative expression> ist der Knoten <state expression>. In der konkreten Syntax wird er mit dem Schlüsselwort **state** notiert. Die Interpretation dieses Knotens ermittelt den Namen des aktuellen Zustands (bei verschachtelten Zuständen den Namen des innersten Zustands).

<state expression> ::

6.11.3 Weitere Ausdrücke

Neben den unären und binären Operatoren erlaubt die Regel <expression> eine Reihe weiterer Alternativen, die im folgenden vorgestellt werden.

Um einen neuen Agenten zu erzeugen, gibt es in SDL die **create**-Aktion. Um dann den Pid-Wert des neuen Agent zu erhalten, muss man den Ausdruck <offspring expression> auswerten. Als Kurznotation für die Erzeugung eines Agenten mit sofortigem Zugriff auf **offspring** gibt es in SDL-2000 den Knoten <create expression>:

<create expression> :: <create body>

Der Knoten <create body> ist der gleiche, wie er auch in der **create**-Aktion angegeben würde. Die Grammatikregeln für <create body> sind nicht Bestandteil der Datentypgrammatik und deshalb hier nicht dokumentiert.

Beispiel 31. Die Aktion

```
task child := create CallHandler(peer);
```

ist gleichbedeutend[20] mit den Aktionen

```
create CallHandler(peer);
task child := offspring;
```

20. Das Transformationsmodell führt für jeden Knoten <create expression> eine anonyme Variable ein, die zunächst den Wert von offspring aufnimmt. In diesem Beispiel kann aber auf die anonyme Variable verzichtet werden, ohne die Bedeutung des ersetzten Fragments zu verändern.

Der Aufruf einer Prozedur wird durch den Knoten <value returning procedure call> repräsentiert:

```
<value returning procedure call> =
        <procedure call body>
    |   <remote procedure call body>
```

In der konkreten Syntax wird der Prozeduraufruf durch das Schlüsselwort **call** eingeleitet, gefolgt von Prozedurbezeichner und aktuellen Parametern[21]. Für den Fall eines Rufs einer entfernten Prozedur kann zusätzlich der Empfänger des Prozedurrufs angegeben werden. Ob es sich bei einem Aufruf um den einer lokalen oder einer entfernten Prozedur handelt, wird in der konkreten Syntax anhand des Prozedurbezeichners entschieden.

Die Grammatikregeln für <procedure call body> und <remote procedure call body> sind nicht Bestandteil der Datentypgrammatik und deshalb hier nicht angegeben; statt dessen zeigt das folgende Beispiel eine Anwendung von werte-liefernden Prozedurrufen.

Beispiel 32. Gegeben sei die (externe) Prozedur

```
procedure call CheckPeer(in Peer peer)->Boolean;external;
```

Diese Prozedur kann in einer Decision-Anweisung wie folgt verwendet werden:

```
decision CheckPeer(peer);
(True): /* Aktionen für den Erfolgsfall */;
(False): /* Aktionen für den Fehlerfall */;
enddecision;
```

Der Knoten <range check expression> erlaubt die Überprüfung, ob ein Ausdruck Element einer eingeschränkten Sorte ist:

```
<range check expression> ::
        <expression> { <range check expression gen identifier> | <sort> }

<range check expression gen identifier> :: <identifier> <constraint>
```

Der Knoten <expression> bezeichnet dabei den zu testenden Ausdruck und der Knoten <sort> beziehungsweise <range check expression gen identifier> die Einschränkung. Falls der Knoten <sort> angegeben ist, wird der Ausdruck durch ein Transformationsmodell ersetzt: Falls <sort> eine Syntype-Sorte bezeichnet, so wird <sort> durch den Knoten <constraint> aus der Syntype-Definition ersetzt. Ansonsten wird der ganze Ausdruck durch den Wert True ersetzt.

In der konkreten Syntax wird der Ausdruck und die Einschränkung durch die Schlüsselwörter **in type** getrennt.

Beispiel 33. Um zu testen, ob eine Zahl i natürlich ist, kann man den Ausdruck

```
i in type Natural
```

verwenden.

Mit dem Knoten <equality expression> wird ein Vergleich zwischen zwei Ausdrücken repräsentiert. In der konkreten Syntax werden die Teilausdrücke durch das Gleichheitszeichen („=“) oder das Ungleichheitszeichen („/=“) getrennt:

```
<equality expression> ::
        <expression> { <equals sign> | <not equals sign> } <expression>
```

21. In SDL-2000 ist das Schlüsselwort **call** optional, sofern keine Mehrdeutigkeit mit Operatorrufen oder Indexzugriffen besteht.

Dieser Ausdruck gehört nicht zu den binären Operatoren, da seine Semantik nicht durch Transformation in einen Operationsruf bestimmt wird. Stattdessen wird zur Interpretation des Vergleichsausdrucks bestimmt, ob die Argumente Objekte oder Werte sind. Zum Vergleich von Objekten werden die Referenzen verglichen (Test auf Identität). Bei Werten wird der Operator equals gerufen, der für jeden Datentyp definiert ist (Test auf Wertegleichheit).

6.12 Variablendefinition und Zuweisungen

Variablen repräsentieren den erweiterten Zustand von Agenten. Sie können in Agenten selbst, in verschachtelten Zuständen, in Prozeduren und in Operationen definiert werden. Der Knoten <variable definition> repräsentiert die Definition mehrerer Variablen:

<variable definition> ::[**exported**] <variables of sort>+

Falls das Schlüsselwort **exported** angegeben ist, sind die Variablen exportierte Variablen. Andere Agenten können auf einen Wert der Variablen über den Ausdruck <import expression> zugreifen. Mehrere Variablen der gleichen Sorte können in einem Knoten <variables of sort> definiert werden:

<variables of sort> ::
{<variables of sort gen name>}+ <sort> [<constant expression>]

Der Kindknoten <sort> bezeichnet die Sorte der Variablen. Falls der Knoten <constant expression> angegeben ist, werden die Variablen mit diesem Ausdruck initialisiert. Falls der Knoten nicht angegeben ist, aber die Typdefinition einen Knoten <default initialization> besitzt, werden die Variablen mit diesem Wert initialisiert. Ansonsten sind die Variablen initial „undefined“. Ein Zugriff auf eine solche Variable liefert die Ausnahme UndefinedVariable.

Die eigentlichen Variablennamen erscheinen im Knoten <variables of sort gen name>[22]:

<variables of sort gen name> :: <name> [<identifier>]

Der Kindknoten <name> bezeichnet den Namen der Variable. Der Knoten <identifier> bezeichnet den Namen, unter dem die Variable exportiert wurde. Dieser Bezeichner darf nur angegeben werden, wenn die Variable eine exportierte Variable ist.

In der konkreten Syntax wird eine Variablendeklaration durch das Schlüsselwort **dcl** eingeleitet. Die Variablen werden dabei durch Kommata getrennt. Der Knoten <constant expression> steht hinter einem Gleichheitszeichen. Der exportierte Name erscheint hinter dem Schlüsselwort **as**.

Beispiel 34. Unter Ausnutzung aller optionalen Teile der Grammatik kann eine Variablendefinition wie folgt lauten

dcl exported a, b Integer = 4, c **as** d Charstring;

Diese Definition führt drei exportierte Variablen a, b und c ein. a und b sind vom Typ Integer, werden mit 4 initialisiert und unter ihrem Namen exportiert, c ist vom Typ Charstring, initial „undefined“ und wird unter dem Bezeichner d exportiert (für den es eine Deklaration einer entfernten Variable geben muss).

Variablen erhalten ihre Werte durch Initialisierung und Zuweisung. Ein Variablenzugriff liefert den letzten zugewiesenen Wert. Eine Zuweisung wird durch den Knoten <assignment> repräsentiert, unter Angabe der Variable und ihres neuen Werts.

<assignment> ::<variable> <expression>

22. Der Namen dieser Grammatikregel wurde automatisch erzeugt, als aus der konkreten Syntax die abstrakte Syntax 0 generiert wurde.

In der konkreten Syntax werden Variable und Wert durch das Zuweisungszeichen („:=“) getrennt.

Die Regel <variable> ist die Vereinigung von drei Alternativen:

<variable>=<identifier> | <indexed variable> | <field variable>

Die Angabe von <identifier> als Variable bezieht sich auf eine Variable, die in einem sichtbaren Knoten <variable definition> definiert wurde. Die beiden anderen Formen dienen der Zuweisung von Feldwerten.

Die Zuweisung eines Wertes an eine indizierte Variable wird durch den Knoten <indexed variable> repräsentiert, und zwar durch Angabe der Variable und der Indizes:

<indexed variable>:: <variable> [<actual parameter>]+

In der konkreten Syntax werden die aktuellen Parameter in eckige oder runde Klammern geschrieben. Wenn eine Zuweisung den Knoten <indexed variable> enthält, so ist das die Kurznotation für den Aufruf des Operators Modify.

Beispiel 35. Die Zuweisung

```
task values[7] := values[6] + 1;
```

ist eine Kurznotation für die Zuweisung

```
task values := Modify(values, 7, "+"(Extract(variable, 6), 1));
```

In dieser Zuweisung wurden neben der Transformation für <indexed variable> auch die Transformationen für <indexed primary> und <binary expression> ausgeführt.

Zum Zuweisung an ein Feld eines Strukturtyps wird der Knoten <field variable> verwendet:

<field variable>:: <variable> <field name>

In der konkreten Syntax wird die Variable vom Feldnamen durch ein Ausrufezeichen oder einen Punkt getrennt. Der Knoten <field variable> ist eine Kurznotation für den Aufruf des entsprechenden Modify-Operators (siehe Abschnitt 6.5).

Beispiel 36. Die Zuweisung

```
task person.age := 21;
```

ist eine Kurznotation für die Zuweisung

```
task person := person.ageModify(21);
```

6.13 Das Paket predefined

Eine Reihe von Datentypen sind in SDL vordefiniert. Die Semantik ihrer Operationen ist nicht durch Operatordefinitionen festgelegt, sondern in der informalen Semantikdefinition durch Übernahme der Axiome von SDL-92. Diese Art der Darstellung sollte die Rückwärtskompatibilität mit SDL-92 verbessern und Fehler vermeiden, die bei Umformulierung der Axiome in englischen Text unterlaufen könnten. In der formalen Semantikdefinition sind diese Operatoren durch Funktionen der dynamischen Datentypsemantik definiert.

Tabelle 4 gibt eine Übersicht über die vordefinierten Datentypen, ihre Literale und Operationen. Die Notation <T> deutet einen parametrisierten Typ an. Eine genaue Definition der Literale und der Operatorsignaturen befindet sich in [Z.100-00, annex D]

Tabelle 4: Vordefinierte Datentypen

Name	Literale	Operationen
Boolean	True, False	**and**, **or**, **not**, **xor**, =>
Character	Steuerzeichen(NUL, SOH, ...), ASCII-Zeichen in Apostrophen ('a', ...)	chr, num, <, <=, >, >=
String<Element>	–	emptystring, mkstring, Make, length, first, last, //, Extract, Modify, substring, remove,
CharacterString = String<Character>	Zeichenfolge in Apostrophen	wie String
Integer	Folgen von Dezimalziffern	+, -, *, /, **mod**, **rem**, <, <=, >, >=, power,
Natural (Integer-Syntype)	wie Integer	wie Integer
Real	Dezimalziffern, optional mit Nachkommastellen	+, -, *, /, <, <=, >, >=, float, fix
Array<Index, Wert>	–	Make, Modify, Extract
Vector<Wert, Maxindex) (Array-Syntype)		
Powerset<Element>	–	empty, **in**, incl, del, <, <=, >, >=, **and**, **or**, length, take
Duration	wie Real	+, -, *, /, >, >=, <, <=
Time	wie Real	<, <=, >, >=, +, -
Bag<Element>	–	wie Powerset
Bit	0, 1	wie Boolean, num, bit
Bitstring = String<Bit>	Binärstring: '001001'B Hexadezimalstring: 'A0B2'H	wie String, **not**, **and**, **or**, **xor**, =>, num, bitstring, octet
Octet = Bitstring size (8)	Wie Bitstring	Wie Bitstring
OctetString = String<Octet>	Wie Bitstring, Zahl der Bits muss durch 8 teilbar sein	Wie String, Octet, Bitstring

Neben den vordefinierten Typen werden im Paket Predefined folgende Ausnahmen vordefiniert:

- OutOfRange (ein Syntype-Test ist fehlgeschlagen),
- InvalidReference (eine Null-Referenz wurde dereferenziert),
- NoMatchingAnswer (in einer Decision-Anweisung gab es keinen passenden Antwortzweig),
- UndefinedVariable (auf eine Variable, die „undefined" ist, wurde zugegriffen),
- UndefinedField (auf ein Feld, dass „undefined" ist, wurde zugegriffen),
- InvalidIndex (der Index ist für das Feld nicht gültig),

- DivisionByZero (Division durch 0) und
- Empty (ein **any**-Ausdruck für eine leere Sorte wurde berechnet).

6.14 Fazit

Das Datentypmodell von SDL-2000 stellt, auf semantischer Ebene, einen vollständigen Bruch mit früheren Sprachversionen dar; auf der syntaktischen Ebene wurden Konstrukte soweit möglich beibehalten, um die Zahl von notwendigen Änderungen an SDL-Programmen zu minimieren.

Kern der Änderungen ist die Einführung eines objekt-orientierten Datentypkonzepts und die Fundierung des Datentypsystems auf der gemeinsamen semantischen Grundlage der gesamten SDL-Sprachdefinition, dem ASM-Kalkül.

7 Definition der statischen Datentypsemantik

Auf der Basis der abstrakten Syntax 0 (AS0) wird die statische Semantik von SDL definiert. Ein SDL-Programm wird in einen Baum der AS0 umgewandelt. Die Korrektheit des Programms wird mit Hilfe von Bedingungen überprüft, die für Knoten der AS0 definiert sind. Der Baum wird mit Hilfe von Transformationsregeln einer Reihe von Transformationen unterzogen, und schließlich auf einen Baum der abstrakten Syntax 1 (AS1) abgebildet. Für den AS1-Baum werden wiederum Korrektheitsbedingungen überprüft. Sofern der AS1-Baum diese Bedingungen erfüllt, ist er das Ergebnis der statischen Semantikanalyse.

In diesem Kapitel wird an ausgewählten Beispielen illustriert, wie diese verschiedenen Aspekte formalisiert wurden und die Formeln zusammenspielen, um die Semantik des SDL-Programms zu definieren. Für jeden Aspekt werden Notationen vorgestellt, mit denen die Formeln für diesen Aspekt notiert werden. Zur Illustration dieser Notationen wurden aus der statischen Semantikdefinition prägnante Beispiele ausgewählt, die den Formalismus illustrieren, ohne den Leser dieser Arbeit mit Details zu überfluten.

7.1 Die Konstruktion der Abstrakten Syntax 0

Die abstrakte Syntax 0 entstand ursprünglich automatisch aus der konkreten Syntax, durch Anwendung des in [Pie00] angegebenen Verfahrens. Dabei wurden Hilfssymbole der konkreten Syntax in Hilfssymbole der abstrakten umgewandelt. Terminalsymbole der konkreten Syntax wurden entfernt, sofern sie keine wesentliche Information tragen, sondern lediglich der Gruppierung oder Kenntlichmachung des Hilfssymbols dienen.

In der abstrakten Syntax gibt es zwei Arten von Regeln: Tupelregeln und Alternativregeln. In Tupelregeln besteht die rechte Seite aus einer Folge von Hilfssymbolen, die von der linken Seite durch **::** abgetrennt wird. In Alternativregeln besteht die rechte Seite aus alternativen Hilfssymbolen, die voneinander durch | und von der linken Seite durch = getrennt sind.

Beispiel 37. In der konkreten Syntax folgt eine Variablendefinition der Regel

<variable definition> ::=

dcl [**exported**] <variables of sort> {, <variables of sort> }* <end>

Daraus ergibt sich in der abstrakten Syntax die Regel

<variable definition> ::[**exported**] <variables of sort>+

Da die Regel der konkreten Syntax nur eine einzige Alternative enthält, ergibt sich in der abstrakten Syntax eine Tupelregel. Das Schlüsselwort **dcl** und das Hilfssymbol <end> werden in der abstrakten Syntax weggelassen, da sie lediglich zur Abgrenzung der Variablendefinition von anderen Definitionen dienen. Das Schlüsselwort **exported** wird in die abstrakte Syntax übernommen, da die Anwesenheit oder Abwesenheit dieses Schlüsselworts zusätzlich Information enthält. Die Kommata, die die einzelnen <variables of sort> separieren, werden wiederum als nicht essentielle Trennzeichen verworfen; damit enthält die Variablendefinition im Inneren nur die Folge <variables of sort> <variables of sort>*; das wird zu <variables of sort>+ zusammengefasst.

7.2 Beispielprädikate für die Abstrakte Syntax 0

Nach Konstruktion der abstrakten Syntax 0 müssen die Bedingungen für die Wohlgeformtheit der abstrakten Syntax überprüft werden. Diese Bedingungen werden in der informalen Seman-

tikdefinition in Form englischer Sätze notiert. Zur Formalisierung wurde für jeden derartigen Satz ein Prädikat angegeben.

Beispiel 38. Virtuelle Operationen oder Operationen mit virtuellen Argumenten müssen stets Methoden sein. Diese Forderung ist auf Englisch wie folgt formuliert:

If <operation signature> is contained in an <operator list>, then the <operation signature> must not contain <virtuality> or <argument virtuality>.

Zur Formalisierung dieser Regel wurde folgendes Prädikat definiert:

$\forall os \in$ <*operation signature*> :
$os.parentAS0 \in$ <*operator list*> $\Rightarrow (\neg \exists v \in$ <*virtuality*> $\cup$ <*argument virtuality*>: *isAncestorAS0(os, v)*)

Hier wird eine Aussage über alle Elemente os der Domäne <operation signature> getroffen. Der Ausdruck os.parentAS0 ist gleichbedeutend mit dem Ausdruck parentAS0(os). In diesem Prädikat wird also Gebrauch von den Hilfsfunktionen parentAS0 und isAncestorAS0 gemacht. Die Funktion parentAS0 ermittelt dabei den unmittelbaren Elternknoten eines Knotens. Die Funktion isAncestorAS0 bestimmt, ob ein Knoten direkter oder indirekter Elternknoten eines anderen Knoten ist.
Da die Produktionsregel für <operator list> unmittelbarer Elternknoten von Knoten des Typs <operation signature> ist, reicht es, die Voraussetzung der Implikation so zu formulieren, dass der Elternknoten von os aus der Domäne <operator list> stammt.
Die Folgerung der Implikation besagt, dass es keine Knoten der Typen <virtuality> oder <argument virtuality> gibt, die ihrerseits als (indirekten) Elternknoten os besitzen.

7.3 Beispiel-Transformationen

Aus dem Baum der abstrakten Syntax 0 wird durch Transformationsschritte wiederum ein solcher Baum erzeugt. Die nötigen Transformationsschritte werden dabei von der informalen Semantikdefinition vorgegeben, ihre Reihenfolge jedoch nicht – dies ist Aufgabe der formalen Semantikdefinition. Zur Definition der Reihenfolge der Transformationsschritte wurde jeder Transformationsregel eine Nummer zugeordnet, sodass die Transformationen in der aufsteigenden Reihenfolge dieser Nummern ausgeführt werden. Haben zwei Transformationsregeln die gleiche Nummer, so können die Regeln in beliebiger Reihenfolge ausgeführt werden.

Ob die Zuordnung dieser Nummern tatsächlich in allen Fällen die gewünschte Semantik von SDL widerspiegelt, ist gegenwärtig noch unbekannt. Zur Definition der Reihenfolge wurde von der Reihenfolge in SDL-92 ausgegangen, wobei berücksichtigt werden musste, dass manche Transformationen in SDL-2000 gar nicht mehr oder nicht mehr in der gleichen Art und Weise benötigt werden, dafür aber neue Transformationsschritte eingeführt wurden.

Sollte sich an konkreten Spezifikationen herausstellen, dass die Transformationen nicht in der gewünschten Reihenfolge ausgeführt werden (also nicht das in der informalen Definition beabsichtigte Ergebnis haben), so muss die formale Definition korrigiert werden. Dabei kann sich auch ergeben, dass die informalen Transformationsregeln widersprüchlich sind, also beispielsweise zwei Transformationsschritte davon ausgehen, dass der jeweils andere Schritt zuvor erfolgt ist. In diesem Fall muss die informale Sprachdefinition korrigiert werden.

In den folgenden Abschnitten werden einige typische Beispiele für Transformationen angegeben. Sie sind Repräsentanten ganzer Klassen von Transformationen, die folgendermaßen charakterisiert werden können:

1. Normalisierung durch syntaktisches Aufblähen
   SDL bietet eine Reihe von Kurzschreibweisen, deren Semantik dem Leser intuitiv offensichtlich ist. Diese Kurzschreibweisen werden in der abstrakten Syntax 1 nicht mehr repräsentiert, da die spezifische Form der Notation nicht wesentlich zur Semantik des Systems

beiträgt, sondern „lediglich“ der Lesbarkeit dient. Deshalb werden die Kurznotationen expandiert, was in der Regel bedeutet, dass der Baum der abstrakten Syntax vergrößert wird.

2. Normalisierung durch Umstellung
   Andere Kurznotationen in SDL sind nicht wesentlich kürzer als die transformierte Version, sondern deshalb lesbarer, weil sie an Notationen aus anderen Sprachen angelehnt sind. Hier besteht die Transformation im Wesentlichen in einer Umstellung der Reihenfolge der Syntaxelemente.
3. Semantikdefinition durch Modelle
   Verschiedene Konstrukte von SDL spiegeln sich in der abstrakten Syntax gar nicht mehr wider. Die Semantik dieser Konstrukte wird ausschließlich durch Transformationsmodelle definiert. Die Formulierung solcher Modelle ist naturgemäß recht aufwändig, da komplizierte Konstrukte durch einfachere ersetzt werden müssen. Für SDL-92 war eine große Zahl solcher Modelle definiert. Bei der Definition von SDL-2000 wurde jedes dieser Modelle darauf untersucht, ob es nicht einfacher ist, dem Konstrukt eine „direkte“, also dynamische Semantik zu geben. Diese Untersuchungen wurden für alle SDL-Konstrukte durchgeführt (also nicht nur für die der Datentypsemantik); dabei wurde dann beispielsweise für Typvererbung sowie für *enabling conditions* das vorher verwendete Transformationsmodell durch eine direkte Semantik ersetzt.

7.3.1 Umformung von Variablendefinitionen

In einer Variablendefinition können mehrere Variablen definiert werden. Beispielsweise werden durch die Definition

```
dcl a,b Integer, c Time;
```

drei Variablen definiert. In der informalen Semantikdefinition gibt es nun die Regel

> *A <variable definition> that defines multiple variables is a shorthand for a sequence of <variable definition>s, each defining one variable.*

Die Beispieldefinition muss also in die Definitionen

```
dcl a Integer;
dcl b Integer;
dcl c Time;
```

transformiert werden. Diese Regel ist also ein Beispiel für die Normalisierung durch Aufblähen: Die Variablendefinitionen werden in eine Normalform gebracht (eine Variable pro Definition). Dabei wird die Spezifikation größer, ohne dass ein Leser einen wesentlichen Wandel in der Bedeutung der Spezifikation ausmachen würde.

Dieser Transformationsschritt wird durch die folgenden Regeln definiert:

< <variable definition>(*exp, < v > ⌢ rest)* > **provided** rest ≠ *empty* =1=>
  < <variable definition> (exp, < v >), <variable definition> (exp, rest) >

< <variables of sort> (< v > ∩ rest, sort, expr) > **provided** rest ≠ *empty* =1=>
  < <variables of sort> (< v >, sort, expr), <variables of sort> (rest, sort, expr) >

In dieser Transformation sind zwei Regeln nötig, weil es zwei Arten von Transformationen gibt. Zum einen kann ein Knoten <variable definition> mehrere Knoten <variables of sort> enthalten:

<variable definition> ::[**exported**] <variables of sort>+

Zum anderen kann ein Knoten <variables of sort> mehrere Variablen definieren (durch die Knoten <variables of sort gen name>):

<variables of sort> ::
{<variables of sort gen name>}+ <sort> [<constant expression>]

Die erste der beiden Regel führt die Expansion des Knotens <variable definition> aus. Voraussetzung der Transformation ist eine Liste (durch den Operator < elemente > dargestellt), die einen Knoten <variable definition> enthält. Dieser Knoten muss als zweiten Kindknoten eine wenigstens ein-elementige Liste enthalten (mit dem Element v). Zusätzlich wird verlangt, dass auch die Restliste (rest) nicht leer ist – damit muss die Liste <variables of sort> also wenigstens zwei Elemente enthalten.

Ein Fragment des Syntaxbaums, das diese Voraussetzungen erfüllt, wird nun im Transformationsschritt 1 durch die rechte Seite der Transformationsregel ersetzt, also durch

< <variable definition> (exp, < v >), <variable definition> (exp, rest) >

Der Ergebnisknoten ist eine Liste aus zwei Elementen, die als erstes Element einen Knoten <variable definition> enthält, der nur die Variablen v definiert, sowie einen Knoten <variable definition>, der alle weiteren Variablen definiert. Sollte die Restliste nun immer noch mehrere Elemente enthalten, wird dieser Transformationsschritt erneut angewendet, bis schließlich alle Knoten <variable definition> im Baum genau einen Kindknoten <variables of sort> besitzen. Falls die Originalvariablendefinition eine exportierte Variable war, sind auch alle entstehenden Variablen exportierte Variablen.

Die zweite Regel funktioniert analog zur ersten. Sie ersetzt einen Knoten <variables of sort>, der mehrere Knoten <variables of sort gen name> enthält, durch mehrere Knoten <variables of sort>. Auch diese Transformation wird in Schritt 1 ausgeführt.

Damit ist es möglich, dass nach Anwendung der zweiten Regel wiederum die Voraussetzungen für die Anwendung der ersten Regel gegeben sind. Man kann sich leicht überlegen, dass dieser Algorithmus der Anwendung von beiden Regeln

1. terminiert und
2. nicht von der Reihenfolge der einzelnen Transformationsschritte abhängt.

7.3.2 Auflösung von Infix-Operatoren

Wie in der Mathematik üblich und auch in anderen Sprachen weit verbreitet, kann man in SDL binäre Operatoren in Infix- und unäre Operatoren in Präfixschreibweise notieren. In der abstrakten Syntax 1 nehmen sie jedoch keine Sonderstellung mehr ein. Dort ist ein Operationsruf definiert durch den Knoten

Operation-application::*Operation-identifier* [*Expression*]*

Ein Operationsruf besteht also aus einem Operationsbezeichner und einer Liste von Ausdrücken (von denen jeder optional ist)- Der Operatorruf 2+4 soll dabei behandelt werden wie der Operatorruf "+"(2, 4). Diese Transformation wird in der informalen Semantik durch die folgende Regel beschrieben:

An expression of the form

<expression> <infix operation name> <expression>

is derived syntax for

<quotation mark> <infix operation name> <quotation mark> (<expression>, <expression>)

where <quotation mark> <infix operation name> <quotation mark> represents an Operation-name.

Der transformierte Operatorruf ist nicht wesentlich größer als der Originalausdruck. Die Kurznotation binärer Ausdrücke ist vor allem deshalb Teil der Sprache, weil sie auch in anderen Sprachen üblich ist und vielen Anwendern nahe liegend erscheint.

Zur Formalisierung dieser Regel wurden die folgenden Transformationsregeln definiert:

<binary expression>(x,<implies sign>,y) =8=> <operator application>("=>",<x,y>)
<binary expression>(x,**or**,y) =8=> <operator application>("or",<x,y>)
<binary expression>(x,**xor**,y) =8=> <operator application>("xor",<x,y>)
<binary expression>(x,**and**,y) =8=> <operator application>("and",<x,y>)
<binary expression>(x,<greater than sign>,y) =8=> <operator application>(">",<x,y>)
<binary expression>(x,<greater than or equals sign>,y) =8=> <operator application>(">=",<x,y>)
<binary expression>(x,<less than sign>,y) =8=> <operator application>("<",<x,y>)
<binary expression>(x,<less than or equals sign>,y) =8=> <operator application>("<=",<x,y>)
<binary expression>(x,**in**,y) =8=> <operator application>("in",<x,y>)
<binary expression>(x,<plus sign>,y) =8=> <operator application>("+",<x,y>)
<binary expression>(x,<hyphen>,y) =8=> <operator application>("-",<x,y>)
<binary expression>(x,<concatenation sign>,y) =8=> <operator application>("//",<x,y>)
<binary expression>(x,<asterisk>,y) =8=> <operator application>("*",<x,y>)
<binary expression>(x,<solidus>,y) =8=> <operator application>("/",<x,y>)
<binary expression>(x,**mod**,y) =8=> <operator application>("mod",<x,y>)
<binary expression>(x,**rem**,y) =8=> <operator application>("rem",<x,y>)

Jede dieser Regeln behandelt Knoten des Typs <binary expression>. Der erste und dritte Kindknoten dieses Knotens ist für die Transformation nicht relevant und wird unverändert übernommen. Der zweite Kindknoten bestimmt den sich ergebenden Operationsnamen.

Dieses Beispiel demonstriert, dass die Formalisierung einer Transformationsregel nicht immer im Verhältnis 1:1 zum englischen Text erfolgen kann. Im englischen Text wurde eine einzige Regel angegeben, die alle Fälle von <infix operation name> abdeckt. Es wäre also zu erwarten, dass auch die Formalisierung mit einer einzigen Regel auskommt.

Wenngleich die Bedeutung dieser Regel intuitiv klar ist, so ist ihre Formulierung jedoch unpräzise: Falls <infix operation name> ein Schlüsselwort ist (wie etwa **mod**), dann bezieht sich die Regel offenbar auf die Schreibweise des Schlüsselworts. In SDL-2000 gibt es dafür zwei Möglichkeiten (**mod** und **MOD**). Demzufolge ist bei strenger Interpretation nicht klar, ob der Operationsname "mod" oder "MOD" lauten soll. Da beide Schreibweisen des Schlüsselworts sich auf den gleichen Operator beziehen sollen und im Paket predefined die Schreibweisen mit Kleinbuchstaben verwendet wird, ist klar, dass offenbar von der Kleinschreibung des Schlüsselworts ausgegangen wird. Diese Annahme wurde in der formalen Transformationsregel manifestiert.

Neben dieser Mehrdeutigkeit gibt es jedoch einen zweiten Grund, warum sich die informale Regel nicht in eine einzige formale Regel umsetzen lässt: In der abstrakten Syntax 0 bilden einige der Infix-Operatoren eigene Syntaxknoten, so ist beispielsweise der Operator // durch die Grammatikregel

<concatenation sign> :: ()

definiert. Aus dieser Definition geht nicht mehr hervor, wie die Schreibweise des Operators in der konkreten Syntax lautete. Die Transformationsregeln bringen dieses Wissen wieder ein, indem sie alle Varianten für <infix operation name> explizit aufzählen.

7.3.3 Transformation von Operatoren in Prozeduren

Mit SDL-2000 müssen Operatoren durch Angabe eines Algorithmus definiert werden (Ausnahme bilden die Operatoren der vordefinierten Datentypen). Der Semantik algorithmischer Operatoren in SDL-96 folgend, werden Operatoren in SDL-2000 in Prozeduren umgeformt. Die

Abarbeitung eines Operators und einer Prozedur erfolgt in der dynamischen Semantik dann auf die gleiche Weise.

Die Umformung von Operatoren in Prozeduren wird durch die folgende informale Definition festgelegt

> *An <operation definition> is transformed into a <procedure definition> or <procedure diagram> respectively, having anonymous name, having <procedure formal parameters> derived from the <formal operation parameters>, and having a <result> derived from the <operation result>. The <procedure body> is derived from <operation body> if one was present, or, if the <operation definition> contains a <statement list>, the result of this transformation is a <procedure definition> (see 9.4). After the Model of <procedure definition> has been applied, the virtual start inserted by that Model is replaced by a start without <virtuality>.*
>
> *An <operation diagram> is transformed into a <procedure diagram> in a similar manner.*
>
> *The Procedure-definition corresponding to the resultant <procedure definition> or <procedure diagram> is associated with the Operation-signature represented by the <operation signature>.*
>
> *If the <operation definition> or <operation diagram> defines a method, then during the transformation into a <procedure definition>, or <procedure diagram> an initial parameter with <parameter kind> in/out is inserted into <formal operation parameters>, with the argument <sort> being the sort that is defined by the <data type definition> that constitutes the scope unit in which the <operation definition> occurs. The <variable name> in <formal operation parameters> for this inserted parameter is a newly formed anonymous name.*

Beispiel 39. Gegeben sei die Operatordefinition

```
operator fib(n Integer)->Integer
{
    if(n<2)
        return 1;
    else
        return n * fib(n-1);
}
```

Nach Anwendung dieser Transformationsregel entsteht die Prozedurdefinition (in textueller Syntax)

```
procedure anon_1(n Integer)->Integer;
{
    if(n<2)
        return 1;
    else
        return n * fib(n-1);
}
```

Für die Operation fib wird zusätzlich vermerkt, dass die Prozedur anon_1 die Operationsdefinition enthält.

Wie aus der Transformationsregel ersichtlich ist, gibt es eine Reihe von Alternativen, die bei der Transformation zu berücksichtigen sind, beispielsweise, ob ein Operationsdiagramm (also grafische Syntax) oder eine Operationsdefinition (textuelle Syntax) zu transformieren sind und ob es sich um einen Operator oder eine Methode handelt.

Diese Alternativen sind in der Formalisierung der Transformationsregel teilweise irrelevant, da in der abstrakten Syntax 0 bereits die Unterscheidung zwischen textueller und grafischer

Syntax hinfällig ist. Die Unterscheidung zwischen Operatoren und Methoden muss hingegen berücksichtigt werden. Die sich daraus ergebende Transformationsregel lautet

```
let nn = newName in
od = <operation definition>(use, <operation heading>(kind, *, *, name, params,
    <operation result>(var, sort), raises), entities, body)
provided od.operatorProcedureName = undefined
=5=>
    od
and
    od.getEntities
=>  od.getEntities ∩
    < <procedure definition>(use,
        <procedure heading>(undefined, undefined, nn, empty, undefined, undefined,
            (if kind = method then
                < <formal procedure parameter>(inout, <parameters of sort>(< thisname >,
                    parentAS0ofKind(od, <data type definition>).identifier0)) >
            else empty endif) ∩
            < <formal procedure parameter>(p.s-<parameter kind>, p.s-<parameters of sort>) |
                p in params >,
            <procedure result>(var, sort), raises),
        entities, makeProcedureBody(body))>
and
    od.operatorProcedureName := nn
```

In dieser Transformation wird von einer Reihe von Hilfsfunktionen Gebrauch gemacht:

- Die Funktion newName liefert anonyme Namen. Die genaue Definition dieser Funktion ist nicht Teil der formalen Semantik (sie ist aus Sicht der Abstract State Machines eine Funktion des Typs **monitored**). Es ist lediglich verlangt, dass sie in jedem Systemzustand einen anderen Wert liefert, der auch verschieden ist von allen Namen, die in der Spezifikation verwendet werden.
- Die Funktion operatorProcedureName ist definiert durch

  ```
  controlled  operatorProcedureName: <operation signature> → <identifier>
  initially ∀ o ∈ <operation signature> : o.operatorProcedureName = undefined
  ```

  Hier handelt es sich um eine ASM-Funktion vom Typ **controlled**. Sie gibt an, ob für eine <operation signature> bereits eine Prozedur definiert wurde. Als Seiteneffekt des Transformationsschritts wird diese Funktion (also die Interpretation des Funktionssymbols) geändert. Da die sonstigen Vorbedingungen der Transformation für alle Knoten des Typs <operation definition> auch nach der Transformation wieder erfüllt sind, würde ohne diese Funktion die Transformationsregel immer wieder ausgeführt.
- Die Funktion getEntities ermittelt die Liste der Definitionen des aktuellen Sichtbarkeitsbereichs.
- Die Funktion makeProcedureBody führt die Umsetzung des Operationskörpers in einen Prozedurkörper durch. Sie ist definiert durch

  ```
  makeProcedureBody(b: <operation body>  <statement list>): <procedure body>  <statement>* =def
      case b of
      | <operation body>(onexc, start, actions) => <procedure body>(onexc, start, actions)
      | <statement list>(*, *) => <compound statement>(b)
      otherwise
          undefined
      endcase
  ```

Die eigentliche Transformationsregel besteht nun aus vier Teilen:

1. Die Vorbedingung ist für alle Knoten <operation definition> erfüllt, die noch nicht transfor-

miert wurden.
2. Die eigentlich Transformation lässt den Knoten <operation definition> unverändert.
3. Der Liste der Definitionen im umgebenden Sichtbarkeitsbereich wird ein Knoten <procedure definition> hinzugefügt, der sich aus dem Knoten <operation definition> im Wesentlichen durch Übernahme aller Kindknoten ergibt. Dabei wird allerdings für Methodendefinitionen ein weiterer impliziter Parameter eingefügt.
4. Der implizite Namen der Prozedur wird in der Funktion operatorProcedureName abgespeichert.

7.3.4 Implizite Operatoren für Strukturtypen

Wie in Abschnitt 6.5 dargestellt, führt die Definition eines Datentyps mit einem Datentypkonstruktor zur Einführung einer Reihe impliziter Operatoren. Die informale Definition dieser Operatoren ist leider inkonsistent. Der Text dieser Beschreibung lautet

A structure definition is derived syntax for the definition of

a) an operator, Make*, to create structures;*

b) methods to modify structures and to access component data items of structures; and

c) methods to test for the presence of optional component data items in structures.

The <arguments> for the Make *operator contains the list of <field sort>s occurring in the field list in the order in which they occur. The result <sort> for the* Make *operator is the sort identifier of the structure sort. The Make operator creates a new structure and associates each field with the result of the corresponding formal parameter. If the actual parameter was omitted in the application of the Make operator, the corresponding field gets no value; that is, it becomes "undefined".*

A <structure definition> introducing a sort named S implies a set of Dynamic-operation-signatures equivalent to the explicit definitions in the following <method list>, for each <field> in its <field list>:

```
virtual field-modify-operation-name ( <field sort> ) -> S;
virtual field-extract-operation-name -> <field sort>;
field-presence-operation-name -> Boolean;
```

where Boolean is the predefined Boolean sort, and <field sort> is the sort of the field.

The name of the implied method to modify a field, field-modify-operation-name, is the field name concatenated with "Modify". The implied method to modify a field associates the field with the result of its argument Expression. When <field sort> was an <anchored sort>, this association takes place only if the dynamic sort of the argument Expression is sort compatible with the <field sort> of this field. Otherwise, the predefined exception UndefinedField *(see Annex D, D.3.16) is raised.*

The name of the implied method to access a field, field-extract-operation-name, is the field name concatenated with "Extract". The method to access a field returns the data item associated with that field. If, during interpretation, a field of a structure is "undefined", then applying the method to access this field to the structure leads to the raise of the predefined exception UndefinedField.

The name of the implied method to test for the presence of a field data item, field-presence-operation-name, is the field name concatenated with "Present". The method to test for the presence of a field data item returns the predefined Boolean value false if this field is "undefined", and the predefined Boolean value true otherwise. A method to test for the presence

of a field data item is only defined if this <field> contained the keyword optional.

Dieser Text definiert nicht nur die Einführung von Operationssignaturen, sondern auch deren Semantik. Diese Semantik wird aber nicht durch Transformation in weitere SDL-Konstrukte erklärt, sondern lediglich durch informalen Text, indem beispielsweise erläutert wird, wie die Semantik von nicht angegebenen Parametern des Operators Make ist.

Tatsächlich lässt sich für diese Operatoren keine Semantik durch Transformation in ein anderes SDL-Konstrukt definieren. Strukturtypen sind in SDL-2000 ein Elementarkonzept, das sich nicht durch andere Konstrukte formalisieren lässt.

Wenngleich der Autor dieser Arbeit während der Entwicklung von SDL-2000 auf diese Inkonsistenz hingewiesen hat, wurde die informale Definition von der SDL-2000-Expertengruppe dennoch als „intuitiv klar“ akzeptiert und die Empfehlung ausgesprochen, in der formalen Semantikdefinition die Unklarheiten zu beseitigen. Daraus ergab sich folgende Transformationsregel:

b = <data type definition body>(*entities*, *s* = <structure definition>(*, *fields*), <operations>(
<operation signatures>(<operator list>(*operators*), <method list>(*methods*)), *refs*, *defs*), *init*)
provided ¬ *b.parentAS0.identifier*$_0$*.implicitSignaturesAdded*

=5=>
(**let** *sort* = b.*parentAS0.identifier*$_0$ **in**
let *newoperators* =
<
<operation signature>(<operation preamble>(*undefined*, **public**), <name>("Make"),
< <argument>(*undefined*, <formal parameter>(**in**,*s1*)) | *s1* **in** *s.fieldSortList*$_0$>,
<result>(*sort*), *empty*)
>
in let *newmethods* =
< <operation signature>(<operation preamble>(**virtual**, **public**),
<name>(*fields.fieldNameList*$_0$[*n*].s-*TOKEN* + "Modify"),
< <argument>(*undefined*, <formal parameter>(**in**, *fields.fieldSortList*$_0$[*n*])) >,
<result>(*sort*), *empty*) | *n* **in** (1..*fields.fieldNameList*$_0$*.length*) > ⌢

< <operation signature>(<operation preamble>(**virtual**, **public**),
<name>(*fields.fieldNameList*$_0$[*n*].s-*TOKEN* + "Extract"),
empty,
<result>(*fields.fieldSortList*$_0$[*n*]), *empty*) | *n* **in** (1..*fields.fieldNameList*$_0$*.length*) > ⌢

< <operation signature>(<operation preamble>(*undefined*, **public**),
<name>(*n*.s-*TOKEN* + "Present"),
empty,
<result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), *empty*)
| *n* **in** *fields.fieldNameList*$_0$: *n.isOptionalField*$_0$ > ⌢
in
<data type definition>(*entities*, *s*, <operations>(<operation signatures>(
<operator list>(*operators* ⌢ *newoperators*), <method list>(*methods* ⌢ *newmethods*),
refs, *defs*), *init*))
endlet)
and
b.parentAS0.identifier$_0$*.implicitSignaturesAdded* := *True*

In dieser Transformationsregel wurden lediglich die Operationssignaturen eingeführt. Die Semantik dieser Operationen ist in der dynamischen Semantik definiert. Ähnlich der Transformation in Abschnitt 7.3.3 ist diese Transformation in folgende Teile gegliedert:

- Das Muster auf der linken Seite führt Transformationen für alle Knoten <data type definition body> durch, deren zweiter Kindknoten vom Typ <structure definition> ist.

- Die Funktion implicitSignaturesAdded kontrolliert, dass die Operationssignaturen nur einmal pro Strukturtyp eingefügt werden. Sie ist definiert als

  controlled *implicitSignaturesAdded*: <identifier> → *BOOLEAN*
  initially ∀ *id* ∈ <identifier>: *id.implicitSignaturesAdded = False*

- Zur Konstruktion der Operationssignaturen werden zwei Variablen gebunden, newoperators und newmethods. Die erste Variable wird stets eine ein-elementige Liste sein. Die Länge der zweiten Liste hängt von der Zahl der Felder ab. Zur Konstruktion der Methoden wird ein Ausdruck der Form *list comprehension* verwendet, der über alle Felder iteriert und für jedes Feld (oder jedes optionale Feld) einen Wert zu der Gesamtliste beiträgt.
- Mit diesen Variablen wird aus der ursprünglichen Datentypdefinition ein neuer Knoten <data type definition> erzeugt, der die gleichen Kindknoten wie der originale Knoten enthält, außer dass zu den Operator- und Methodenlisten die neuen Operationen hinzugefügt werden.
- Da die Transformationsregel nur einmal pro Strukturtyp abgearbeitet werden darf, wird die Funktion implicitSignaturesAdded geändert.

In der dynamischen Semantik sind keine Informationen mehr über die ursprüngliche Strukturtyp vorhanden, da der Knoten <structure definition> der abstrakten Syntax 0 keine Entsprechung in der abstrakten Syntax 1 hat. Damit stellt sich die Frage, wie denn die dynamische Semantik den Operatoren die gewünschte Semantik geben soll: Der nahe liegende Ansatz, die Menge aller Felder eines Strukturobjekts aus der abstrakten Syntax zu ermitteln, lässt sich jedenfalls nicht realisieren.

Wie in Abschnitt 8.6 dargestellt wird, basiert die dynamische Semantik von Strukturen auf der Erkennung spezieller Operationsnamen: Eine Operation, die keine zugeordnete Prozedur hat (siehe Abschnitt 7.3.3), und deren Namen „Make“ lautet, wird dynamisch als Konstruktionsoperator interpretiert.

Dabei stellen sich immer noch einige Fragen:

1. Wie soll die dynamische Semantik erkennen, ob ein Feld "undefined" ist, also keinen Wert hat?
2. Woher kommt in der dynamischen Semantik der Standardwert für Felder, die einen Standardwert besitzen?

Die erste Frage lässt sich durch geeignete Definition von Strukturen in der dynamischen Semantik definieren: Wenn Strukturen als Folgen von Name-Wert-Paaren repräsentiert werden, kann das Fehlen eines Namens als nicht-belegtes Feld interpretiert werden.

Zur Beantwortung der zweiten Fragen wäre es möglich gewesen, die Berechnung des Standardwerts beim Zugriff (als im Extract-Operator) zu realisieren. Dazu wäre eine Änderung der abstrakten Syntax 1 erforderlich gewesen, um festzuhalten, welches Feld welchen Standardwert hat.

Um diese Änderung zu vermeiden, wurden Standardwerte ebenfalls über ein Modell definiert: Hat ein Feld einen Standardwert, und wird eine Strukturinstanz erzeugt, bei der dieser Feldwert nicht angegeben ist, so erfolgt unmittelbar nach der Erzeugung eine Belegung des Felds mit dem Standardwert. Diese Lösungsstrategie wurde als Transformationsmodell in die SDL-Semantik aufgenommen.

7.4 Beispiele für die Konstruktion der Abstrakten Syntax 1

Zur Abbildung der abstrakten Syntax 0 auf die abstrakte Syntax 1 wird eine Funktion Mapping definiert, die durch die Signatur

Mapping: *DefinitionAS0 DefinitionAS1*

deklariert wird. Diese Funktion besteht aus einem einzigen **case**-Ausdruck, der für alle relevanten Muster der abstrakten Syntax 0 einen Ausdruck angibt, der den entsprechenden Knoten der abstrakten Syntax 1 konstruiert.

Für die meisten Konstrukte ist die Abbildung in der informalen Semantik ohne spezielle Erläuterung im englischen Text definiert, da in der Einleitung von [Z.100-00] eine allgemeine Regel aufgestellt wird: Wenn keine speziellen Aussagen getroffen werden, dann wird das Nichtterminal <x> der konkreten Syntax auf die gleichnamige Regel *X* der abstrakten Syntax 1 abgebildet. Ein Beispiel für eine derartige Abbildung wird in Abschnitt 7.4.1 gegeben. Die Abschnitte 7.4.2 und 7.4.3 geben Beispiele für nicht-triviale Abbildungen.

7.4.1 Gleichheit von Ausdrücken

In der abstrakten Syntax 0 ist die Gleichheit und Ungleichheit von Ausdrücken durch die Grammatikregel

```
<equality expression> ::
        <expression> { <equals sign> | <not equals sign> } <expression>
```

definiert. Nach dieser Regel hat der Knoten <equality expression> zwei Formen: Wird zwischen den beiden Ausdrücken der Knoten <equals sign> (in der konkreten Syntax „=“) verwendet, testet der Ausdruck auf Gleichheit. Wird der Knoten <not equals sign> („/=“) verwendet, testet der Ausdruck auf Ungleichheit.

In der abstrakten Syntax 1 lautet die entsprechende Regel

```
Equality-expression   ::   First-operand Second-operand
First-operand         =    Expression
Second-operand        =    Expression
```

Mit dieser Syntax kann nur noch die Gleichheit von Ausdrücken erfasst werden. Die Semantik des Knotens <not equals sign> wird durch ein Transformationsmodell erklärt:

```
<equality expression>(x, <not equals sign>, y)
=8=> <expression gen primary>(not, <equality expression>(x, <equals sign>, y))
```

Mit diesem Modell wird ein Ausdruck, der den Operator „/=“ verwendet, in einen Ausdruck transformiert, der den Operator „=“ verwendet, und das Ergebnis negiert.

Damit muss sich die Abbildung auf die abstrakte Syntax 1 nur noch mit dem Fall <equals sign> beschäftigen. Da die Knoten der abstrakten Syntax genauso heißen wie die der konkreten Syntax, wird im Haupttext von [Z.100-00] keine weitere Regel angegeben. Die Formalisierung der Abbildung lautet

```
| <equality expression>(first, <equals sign>, second)
    => mk-Equality-expression(Mapping(first), Mapping(second))
```

Mit dieser **case**-Alternative der Funktion Mapping wird aus einem Knoten <equality expression> ein Knoten *Equality-expression* erzeugt. Für die beiden Argumentausdrücke wird die Funktion Mapping aufgerufen, die die Entsprechung der Argumente in der abstrakten Syntax 1 ermittelt.

7.4.2 Unterscheidung von Assignment und Assignment Attempt

Zur Umwandlung einer Referenz auf einen Basistyp in eine Referenz auf eine Ableitung dieses Typs (*assignment attempt*) wird in SDL in der konkreten Syntax das Zuweisungszeichen benutzt, genau wie bei einer normalen Zuweisung (siehe Abschnitt 3.4). In der abstrakten Syntax

1 wird zwischen diesen beiden Formen der Zuweisung jedoch unterschieden. Die abstrakte Syntax lautet hierfür

Assignment :: *Variable-identifier Expression*
Assignment-attempt :: *Variable-identifier Expression*

Zur Konstruktion der abstrakten Syntax aus der konkreten wird in der informalen Grammatik die folgende Formulierung verwendet:

> *If the <identifier> has been declared with an object sort and the sort of the <expression> is a (direct or indirect) supersort of the sort of the <identifier>, the <assignment> represents an Assignment-attempt. Otherwise, the <assignment> represents an Assignment.*

Diese Beschreibung wurde durch folgende case-Variante in der Funktion Mapping formalisiert:

```
| <assignment>(var,expr)
    => if isAttempt(var.staticType, expr.staticType) then
           mk-Assignment-attempt(Mapping(var), Mapping(expr))
       else
           mk-Assignment(Mapping(var), Mapping(expr))
       endif
```

In diesem Ausdruck werden eine Reihe von Hilfsfunktionen verwendet:

- Die Funktion staticType ermittelt für einen Ausdruck den statischen (deklarierten) Datentyp.
- Die Funktion isAttempt bestimmt für zwei Typen, ob der eine eine Spezialisierung des anderen ist. In diesem Fall wird ein Knoten vom Typ *Assignment-attempt* konstruiert, ansonsten einer vom Typ *Assignment*.

Beide Formen der Zuweisung besitzen in der abstrakten Syntax 1 zwei Kindknoten. In beiden Fällen ergeben sich diese durch Aufruf der Funktion Mapping für die Variable und den Ausdruck in der abstrakten Syntax 0.

7.4.3 Konstruktion von Object- und Value-Typen

Zur Darstellung von Datentypdefinitionen gibt es in der abstrakten Syntax 1 folgende Konstrukte:

Data-type-definition = *Value-data-type-definition*
| *Object-data-type-definition*
| *Interface-definition*

Value-data-type-definition :: *Sort*
Data-type-identifier
Literal-signature-**set**
Static-operation-signature-**set**
Dynamic-operation-signature-**set**
Data-type-definition-**set**
Syntype-definition-**set**
Exception-definition-**set**

Object-data-type-definition :: *Sort*
Data-type-identifier
Literal-signature-**set**
Static-operation-signature-**set**
Dynamic-operation-signature-**set**
Data-type-definition-**set**
Syntype-definition-**set**
Exception-definition-**set**

Der Knotentyp *Interface-definition* entsteht, der allgemeinen Regel folgend, aus dem Knoten <interface definition>. Schwieriger ist die Erzeugung der Knoten *Value-data-type-definition* und *Object-data-type-definition*. Obwohl es keine gleichnamigen Konstrukte in der konkreten Syntax gibt, wird in der informalen Semantik jedoch auch keine Konstruktionsregel für die abstrakte Syntax angegeben.

Jedoch ist offensichtlich, dass diese Knoten zur Darstellung von Wertetypen und Objekttypen gedacht sind. Auf dieser Interpretation basierend wurde folgende formale Abbildung definiert:

```
| <data type definition>(*, *, <data type heading>(value, name, *, *), base, body) =>
    mk-Value-data-type-definition(Mapping(name), Mapping (base),
        { e ∈ Mapping(body).toSet: e ∈ Literal-signature },
        { e ∈ Mapping(body).toSet: e ∈ Static-operation-signature },
        { e ∈ Mapping(body).toSet: e ∈ Dynamic-operation-signature },
        { e ∈ Mapping(body).toSet: e ∈ Data-type-definition },
        { e ∈ Mapping(body).toSet: e ∈ Syntype-definition },
        { e ∈ Mapping(body).toSet: e ∈ Exception-definition })
| <data type definition>(*, *, <data type heading>(object, name, *, *), base, body) =>
    mk-Object-data-type-definition(Mapping(name), Mapping (base),
        { e ∈ Mapping(body).toSet: e ∈ Literal-signature },
        { e ∈ Mapping(body).toSet: e ∈ Static-operation-signature },
        { e ∈ Mapping(body).toSet: e ∈ Dynamic-operation-signature },
        { e ∈ Mapping(body).toSet: e ∈ Data-type-definition },
        { e ∈ Mapping(body).toSet: e ∈ Syntype-definition },
        { e ∈ Mapping(body).toSet: e ∈ Exception-definition })
| <data type definition body>(entities, ctor, operations, *) =>
        Mapping(entities) ∩ Mapping(ctor) ∩ Mapping(operations)
```

Nach dieser Abbildungsregel wird ein Knoten <data type definition> auf den Knoten *Value-type-definition* abgebildet, wenn im Knoten <data type heading> das erste Element value ist, ansonsten entsteht ein Knoten *Object-type-definition*.

Die verschiedenen Elemente des Körpers der Datentypdefinition werden zunächst über die Regel für <data type definition body> auf eine Liste von Knoten abgebildet. Zur Konstruktion der Datentypdefinition werden dann die Listenelemente je nach Zugehörigkeit zu einem Knotentyp (*Literal-signature*, *Static-operation-signature* usw.) in die Kindknoten der Datentypdefinition eingefügt.

7.5 Beispielprädikate für die Abstrakte Syntax 1

Nachdem die abstrakte Syntax 1 durch die Funktion Mapping konstruiert wurde, müssen wiederum Korrektheitsregeln (*well-formedness rules*) überprüft werden. Wie auch die Regeln für die abstrakte Syntax 0 sind diese Regeln als Prädikate formuliert. Diese Prädikate geben üblicherweise den Text der englischen Formulierung direkt wieder. Zur Illustration wurden zwei typische Beispiele ausgewählt.

7.5.1 Typrichtigkeit von Konstanten in Variableninitialisierungen

Wird eine Variable mit einem Standardwert belegt, wie in der Definition

```
dcl i Integer := <<type Integer>> 0;
```

so muss der Typ der Konstanten der gleiche sein wie der Typ der Variablen. In der informalen Definition ist diese Regel formuliert durch

If the Constant-expression is present, it must be of the same sort as the one denoted by Sort-reference-identifier.

Diese Formulierung bezieht sich dabei auf die folgende Regel der abstrakten Syntax:

Variable-definition :: *Variable-name*
Sort-reference-identifier
[*Constant-expression*]

Zur Formalisierung dieser Regel wurde folgendes Prädikat definiert

$\forall d \in$ *Variable-definition: d.***s-***Constant-expression* $\neq$ *undefined* $\Rightarrow$
d.**s-***Constant-expression*.*staticSort*$_1$= d.s-*Sort-reference-identifier*

Falls der Knoten *Constant-Expression* im Knoten *Variable-definition* weggelassen ist, so liefert die Zugriffsfunktion **s-***Constant-expression* den Wert *undefined.* Anderenfalls verlangt das Prädikat, dass die Funktion *staticSort*$_1$, die für einen Knoten der abstrakten Syntax 1 seine deklarierte Sorte ermittelt, den Bezeichner der Sorte der Variable zurückgibt.

7.6 Fazit

Die Definition der statischen Semantik von SDL besteht aus 5 Teilen: abstrakter Syntax 0 (AS0), Korrektheitsbedingungen, Transformationsregeln, abstrakter Syntax 1 (AS1) und der Abbildung von AS0 auf AS1.

Für jeden dieser Teile wurde eine Notation entwickelt, die von Werkzeugen automatisch verarbeitet werden kann. Die semantische Fundierung aller Teile ist der ASM-Kalkül.

Bei der Formalisierung der informalen Sprachdefinition wurden zahlreiche Probleme entdeckt; dieses Kapitel illustriert einige dieser Probleme und die gewählten Lösungsstrategien.

8 Definition der dynamischen Datentypsemantik

Während die Datentypsemantik von SDL-92 auf dem Kalkül der initialen Algebra basiert, liegt der Datentypsemantik von SDL-2000 das gleiche Kalkül zugrunde wie der gesamten SDL-Semantik, nämlich das der *Abstract State Machines*.

Wie in Abschnitt 5.3.1 dargestellt, wird ein Ablauf eines SDL-Systems durch eine Menge verteilter ASM-Agenten sowie eine halbgeordnete Folge von Systemzuständen repräsentiert. Der Systemzustand wird dabei durch die Programme der Agenten geändert.

Dieser Systemzustand wird durch ein Vokabular von Funktionen und dessen Interpretation beschrieben, wobei natürlich auch die Variablenbelegungen erfasst sein müssen.

Aus der Erfahrung von früheren SDL-Versionen sowie aus der Erfahrung im praktischen SDL-Einsatz gab die Expertengruppe für die formale Semantik von SDL-2000 die Empfehlung, die Datentypsemantik weitestgehend von der Semantik von Agenten zu entkoppeln.

Der Grund für diese Empfehlung war der Wunsch, die Datentypsemantik bei Bedarf durch die Semantik einer anderen Sprache (etwa von C++) austauschen zu können. Dabei wurde allerdings nicht genauer spezifiziert, welche Art von Flexibilität in Bezug auf die Formulierung der Datentypsemantik denn sinnvoll ist.

Zur Erfüllung dieser unscharfen Forderung wurde eine *funktionale* Schnittstelle favorisiert, in der also die eigentlichen Berechnungen durch Funktionen ohne Seiteneffekte erfolgen. Damit kann das Datentypmodell prinzipiell durch ein anderes ersetzt werden, sofern dieses andere Modell ebenfalls funktional ist. In Abschnitt 8.1.2 wird gezeigt, dass diese Forderung leider zu einer komplizierteren und weniger intuitiven Repräsentation von Variablen und deren Werten führt.

Bisher ist die Definition einer alternativen Datentypsemantik nicht erfolgt. Es ist auch zweifelhaft, ob sich eine beliebige andere Datentypsemantik „einfach“ in SDL integrieren lassen würde. Es ist eher wahrscheinlich, dass die Integration einer anderen, bereits formulierten Semantik auf folgende Probleme trifft:

- Eine andere Datentypsemantik wird oft nicht funktional formuliert sein. So ist beispielsweise die naheliegende Idee, die schon auf der Basis des ASM-Kalküls definierte Java-Semantik [BS98] in SDL zu integrieren, schwer zu realisieren, da hier die Berechnungen der virtuellen Java-Maschine durch Zustandsänderungen eines ASM-Agenten repräsentiert werden.
- Selbst wenn eine vorhandene Datentypsemantik funktional ist, kann sie doch nicht einfach in SDL-2000 integriert werden: Die Datentypschnittstelle besitzt eine Reihe spezifischer Funktionen, die eine alternative Semantik ebenfalls realisieren müsste. Gleichzeitig beschränkt sich die dynamische SDL-Semantik auf Rufe genau dieser Funktionen, und verwendet deren Ergebnis zur Aktualisierung des Systemzustands. Eine alternative Semantik kann also Änderungen des Systemzustands nur auf die gleiche Weise bewirken, wie die Datentypsemantik von SDL-2000 das auch tut.

Eine Ausnahmestellung nimmt hierbei die Integration von ASN.1 ein, da die bessere Integration von ASN.1-Datentypen in SDL ein wichtiges Ziel der Neugestaltung der Datentypen war. Hierbei wurde jedoch nicht von dem Mechanismus Gebrauch gemacht, die Datentypsemantik austauschen zu können. Vielmehr sollen ja ASN.1-Datentypen zusammen mit den Standardkonstrukten für Daten von SDL verwendet werden können. Die Integration von ASN.1 wurde statt dessen durch ein Transformationsmodell erreicht [Z.105], sodass ASN.1-Typen die gleiche dynamische Semantik besitzen wie „normale“ SDL-Datentypen. Wenngleich dieses Transformationsmodell nicht formalisiert wurde, so ist es doch wahrscheinlich, dass sich die in Kapitel 7 vorgestellten Mechanismen für die Beschreibung dieser Transformation einsetzen las-

sen. Dies wird insbesondere dadurch vereinfacht, dass ASN.1-Datentypen lediglich Zustand repräsentieren, und [X.680] selbst keine Operationssemantik für solche Datentypen festlegt.

In den folgenden Abschnitten werden die wesentlichen Teile der Datentypsemantik beschrieben. Der Schwerpunkt der Darstellung liegt dabei in der Integration der Datentypsemantik in den „Rest“ der formalen Semantik.

8.1 Entwicklung einer funktionalen Schnittstelle

Um die Datentypsemantik von SDL-2000 zu formalisieren, müssen Antworten auf folgende Fragen gefunden werden:

1. Wie werden Werte von Ausdrücken repräsentiert?
2. Wie wird die Belegung von Variablen repräsentiert?
3. Wie erfolgt die Berechnung von Operatoren?

Die Beantwortung der ersten Frage ist Voraussetzung der Beantwortung der zweiten und dritten Frage: Solange man nicht weiß, was ein Wert ist, kann man auch nicht definieren, wie der Wert einer Variablen ermittelt wird. Genauso wenig kann man definieren, was ein Operatorruf ist, da dieser Werte als aktuelle Argumente haben wird.

8.1.1 Werte

Auf der Basis der Domänenkonstruktionsmöglichkeiten der formalen Semantikdefinition wurde die Domäne Value definiert als Vereinigung aller möglichen Werte, die in SDL auftreten können, also als Vereinigung

$$\begin{aligned} \mathit{VALUE} =_{\mathrm{def}} \; & \mathit{SDLINTEGER} \cup \mathit{SDLBOOLEAN} \cup \mathit{SDLREAL} \cup \mathit{SDLCHARACTER} \cup \mathit{SDLSTRING} \\ & \cup \mathit{PID} \cup \mathit{OBJECT} \cup \mathit{SDLLITERALS} \cup \mathit{SDLSTRUCTURE} \cup \mathit{SDLARRAY} \cup \mathit{SDLPOWERSET} \\ & \cup \mathit{SDLBAG} \cup \mathit{SDLTIME} \cup \mathit{SDLDURATION} \end{aligned}$$

Jede der einzelnen Teildomänen wird in den folgenden Abschnitten definiert. Zunächst waren für die Domänen, die durch vordefinierte Domänen des ASM-Kalküls realisiert wurden (also *INTEGER*, *BOOLEAN*) in der Domäne *VALUE* tatsächlich die vordefinierten Domänen aufgeführt. Es zeigte sich aber, dass mit dieser Repräsentation keine polymorphen Variablen modelliert werden können, da bei Spezialisierung eines vordefinierten Typs (etwa Boolean) in SDL die dynamische Repräsentation keine Typzugehörigkeit mehr erfasst. Deshalb sind die einzelnen Teildomänen in der Regel Tupeldomänen, bei denen die zweite Projektion den dynamischen Typ im SDL-Sinne enthält.

Da eine Reihe von Funktionen der Datentypschnittstelle Parameter vom Typ Value haben, muss auch diese Domäne Teil der Schnittstelle sein – allerdings nur ihr Name, nicht (wie hier angegeben) auch ihre Definition.

8.1.2 Zustände

Mit Festlegung der Domäne *VALUE* kann man nun das Konzept der Variablenbelegung definieren. Dabei müssen folgende Aspekte beachtet werden:

- Variablen mit verschiedenem Bezeichnern können verschiedene Werte haben.
- Variablen mit gleichem Bezeichner können auch verschiedene Werte haben, wenn sie zu verschiedenen SDL-Agenten oder zu verschiedenen Prozeduraufrufen gehören.
- Innerhalb eines Prozessagenten können verschachtelte Agenten sowie gerufene Prozeduren auf die Variablen umliegender Sichtbarkeitsbereiche zugreifen[23].
- Sofern der zugreifende Agent und die zugreifende Prozedur bekannt sind, hat eine

bestimmte Variable zu jedem Zeitpunkt nur einen Wert. Bei Zuweisung ändert sich dieser Wert.

Um begrifflich die Unterscheidung zwischen einer in der abstrakten Syntax deklarierten Variablen und ihren potentiell mehreren zugeordneten Werten zu ermöglichen, sollen folgende Begriffe eingeführt werden:

- Der *Variablenbezeichner* ist die Kennung einer Variablen in der abstrakten Syntax: Jeder Knoten *Variable-definition* der abstrakten Syntax 1 ist genau einem Variablenbezeichner zugeordnet.
- Die *Variable* ist eine konkrete Inkarnation eines Variablenbezeichners, in einem konkreten Kontext (also einem Agenten oder einem Prozedurruf).
- Der *Variablenwert* ist der Wert, den eine Variable in einem bestimmten Systemzustand hat.

Zwischen diesen Begriffen gelten die Relationen, wie sie in Abbildung 11 dargestellt sind:

- Für eine Variablendefinition kann es beliebig viele Variablen geben.
- Eine Variable kann während der Abarbeitung eines Systems beliebig viele Werte haben.

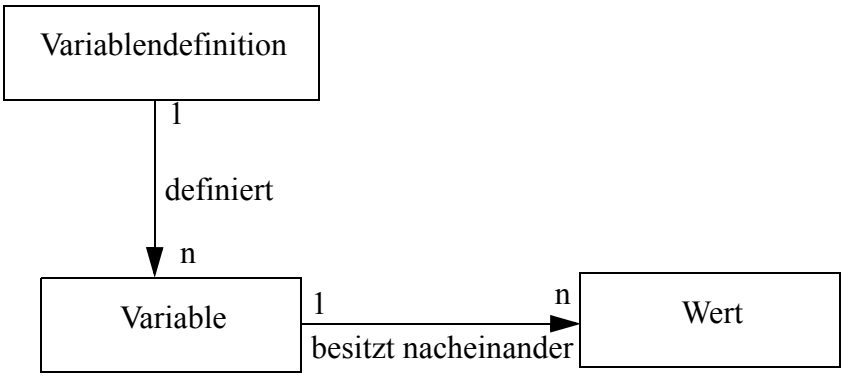

Abbildung 11: Relation zwischen Variablendefinition, Variablen und Werten

Um den Wert einer Variable zu ermitteln, benötigt man die Variablendefinition sowie den „Kontext“, in dem der Variablenzugriff erfolgt. Es liegt nahe, als „Kontext“ den Agenten zu verwenden, der auf die Variable zugreift. Mit dieser Definition lässt sich aber die SDL-Semantik nicht wiedergeben, da beispielsweise innerhalb einer rekursiven Prozedur innerhalb eines Agenten die gleiche Variablendefinition gleichzeitig mehreren Werten zugeordnet sein kann (in verschiedenen der verschachtelten Prozeduraufrufe).

Während der Interpretation eines Agenten werden offenbar dynamisch neue Sätze von Variablen angelegt, beispielsweise jedes Mal, wenn eine Prozedur gerufen wird. Ein solcher Satz von Variablen wird nun in der Datentypsemantik durch die Domäne State beschrieben. Unter Ausnutzung der ASM-Konzepte könnte nun die Zuordnung von Variablen zu Werten relativ einfach durch eine Funktion vom Typ **controlled** erfolgen:

controlled *variableValue*: *STATE* × *IDENTIFIER* → *VALUE*

Leider verletzt diese Definition die Forderung, dass die Datentypsemantik eine funktionale Schnittstelle besitzt: Die Änderung dieser Funktion variableValue ist Aufgabe der Datentypse-

23. Innerhalb eines Blocks können Prozeduren und Prozesse ebenfalls auf die Blockvariablen zugreifen. Allerdings erfolgt dieser Zugriff nicht direkt, sondern durch Signalaustausch. Die Formalisierung des Signalaustauschs ist nicht Aufgabe der Datentypsemantik.

mantik. Jede Änderung von **controlled**-Funktionen ist aber ein Seiteneffekt, der ja für die Datentypsemantik nicht erlaubt sein soll.

Um überhaupt Zustandsänderungen modellieren zu können, muss man annehmen, dass jeder Agent bei einer Zuweisung einer Variablen eine Funktion der Datentypsemantik ruft, und das Ergebnis dieses Funktionsrufs an eine **controlled**-Funktion zuweist. Die Funktion

assign: Variable-identifier × VALUE × STATE → STATE

könnte beispielsweise dazu benutzt werden, Zuweisungen zu realisieren: Als Argumente erhält sie die Variable, den Wert und den alten Zustand, als Ergebnis liefert sie den neuen Zustand. Die eigentliche Repräsentation des Zustands ist dann Aufgabe der Datentypsemantik. Da es sich sicher um einen strukturierten Wert handelt, bei Erzeugung eines Agenten oder bei Aufruf einer Prozedur eine Initialisierungsfunktion, etwa initState, gerufen werden, die beispielsweise auch die Standardwerte für Variablen setzt.

Nun kann bei einer Zuweisung an eine Variable in einer Prozedur die Variable auch vom umliegenden Agenten definiert sein.

Beispiel 40. In textueller Syntax wird hier innerhalb einer Prozedur eine Variable des Agenten geändert.

```
process Agent;

    dcl i Integer;

    procedure Funktion(k Integer);
        dcl m Integer;
        start;
            task m := k; /* lokale Zuweisung */
            task i := k;   /* globale Zuweisung */
            return;
    endprocedure;

    start;
        call Funktion(5);
        stop;

endprocess;
```

Um die beiden Zuweisungen des Beispiels zu realisieren, müsste die Agentensemantik für die beiden Zuweisungen verschiedene Zustandsobjekte übergeben, und dazu also ermitteln, welche Variablen in welchen Zuständen vorhanden sind. Die Interpretation der Zustände ist aber Aufgabe der Datentypsemantik.

Also muss die Agentensemantik stets den *gesamten* Zustand des Agenten (also all seine Variablen) an die Datentypsemantik übergeben, welche dann einen neuen Gesamtzustand zurückgibt.

Der Gesamtzustand ist hierbei der Zustand, auf den verschachtelte Definitionen „direkt“ (also ohne Kommunikation) zugreifen können. Wie in Abbildung 12 dargestellt, ist er hierarchisch aufgebaut:

- Innere Agenten können auf die Variablen äußerer Agenten zugreifen.
- Verschachtelte SDL-Zustände können auf die Variablen umliegender Zustände und auf die des Agenten zugreifen.
- Innere Prozeduren können auf die Variablen ihrer umliegenden Rufer zugreifen. Durch das State-Aggregation-Konstrukt können in einem Agenten mehrere Prozedurstacks bestehen.

Aus diesen Betrachtungen folgt, dass stets der *äußerste* Prozess-Agent den Zustand aller verschachtelten Agenten sowie der Prozeduren verwaltet. Ein Blockagent ist selbst sein äußerster

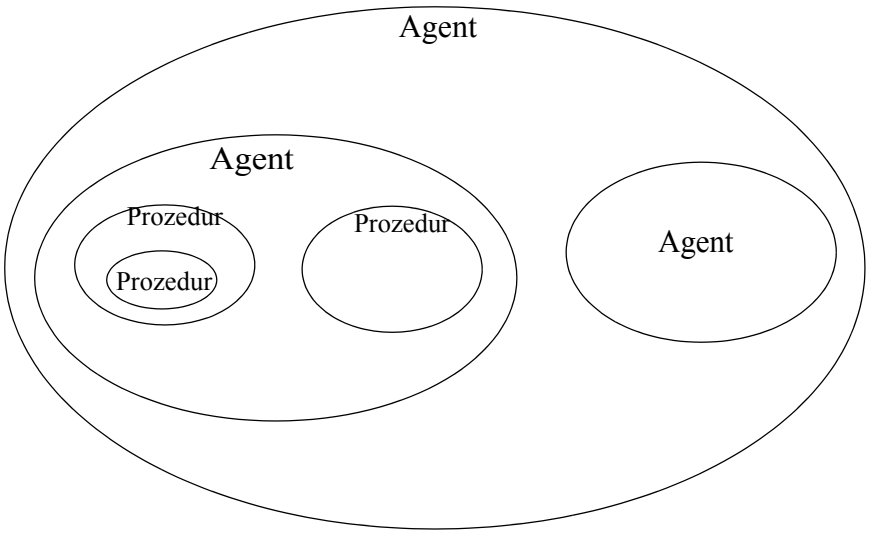

Abbildung 12: Variablenbindungen sind in SDL verschachtelt.

Agent. Die Verschachtelung betrifft dann lediglich Prozeduren, die in dem Block selbst gerufen werden.

Da innerhalb eines solchen Gesamt-Zustands jede Variable mehrfach gebunden sein kann, müssen zur Kennung der Variablen Teile des Zustands identifiziert werden. Um diese Identifikation zu ermöglichen, wurde der Typ *StateID* definiert:

controlled domain *StateID*

Werte dieses Typs werden von der Agentensemantik dynamisch erzeugt, wenn ein neuer Teilzustand benötigt wird[24]. Der Zugriff auf eine Variable benötigt nun also den Variablenbezeichner, den Zustand, sowie die Zustandsidentifikation. Die Variablenzugriffsfunktion kann damit durch folgende Signatur deklariert werden:

eval: Variable-identifier $\times$ *State* $\times$ *StateID* $\rightarrow$ *Value*

Die Änderungsfunktion für Variablen ändert sich damit zu

assign: Variable-identifier $\times$ *Value* $\times$ *State* $\times$ *StateID* $\rightarrow$ *State*

Zusätzlich werden noch einige Initialisierungsfunktionen benötigt, mit denen der Zustand um initiale Variablenbindungen angereichert werden kann:

Declaration $=_{DEF}$ *Procedure-formal-parameter* $\cup$ *Variable-definition*

initAgentState: [*State*] $\times$ *StateID* $\times$ [*StateID*] $\times$ *Declaration*-set$\rightarrow$ *State*

initProcedureState: *State* $\times$ *StateID* $\times$ *StateID* $\times$ *Declaration*-set $\times$
*ValueOrIdentifier** $\times$ *Value** $\times$ *Variable-identifier**$\rightarrow$ *State*

Zur Initialisierung eines Agentenzustands wird optional der alte Agentenzustand, die neue Zustandidentifikation, optional die Identifikation des umliegenden Zustands sowie die Liste der zu deklarierenden Variablen benötigt. Bei der Initialisierung des Zustands einer Prozedur ist die Angabe des umliegenden Zustands nicht optional, zusätzlich werden die Parameternamen und Werte benötigt.

24. Auch die Erzeugung eines neuen Werts in einer **controlled**-Domäne ist eine Operation mit Seiteneffekt, also nicht-funktional.

8.1.3 Ausnahmen

Die Operationen der Datentypsemantik können neben einer erfolgreichen Berechnung eines Ergebnisses auch Ausnahmen produzieren. Falls eine Operation eine Ausnahme liefert, ändert sich der Systemzustand nicht.

Zur Repräsentation von Ausnahmen in der Datentypschnittstelle wurde die Domäne Exception definiert:

$EXCEPTION =_{def} Identifier$

Auf die Repräsentation von Parametern der Ausnahme konnte verzichtet werden, da in der Datentypsemantik nur die vordefinierten Ausnahmen auftreten können, und diese sämtlich ohne Parameter auskommen.[25] Dies bedeutet natürlich eine Einschränkung für mögliche alternative Datentypsemantiken (etwa von C++): Falls sie Operatoren realisieren wollen, die parameterbehaftete Ausnahmen auslösen, muss die Schnittstelle der Datentypsemantik angepasst werden.

Berechnungen der Datentypsemantik geben nun entweder einen neuen Zustand oder eine Ausnahme zurück. Dieser Rückgabetyp wird durch die Domäne *StateOrException* definiert. Analog kann die Domäne *ValueOrException* wahlweise Werte oder Ausnahmen repräsentieren:

$STATEOREXCEPTION =_{def} STATE \cup EXCEPTION$

$VALUEOREXCEPTION =_{def} VALUE \cup EXCEPTION$

Da auch Zuweisungen Ausnahmen auslösen können, ändert sich die Signatur der Funktion assign in die endgültige Form

$assign: Variable\text{-}identifier \times VALUE \times STATE \times STATEID \rightarrow STATEORECEPTION$

8.1.4 Berechnung von Operatoren

Wie in Abschnitt 7.3.3 bereits dargestellt, werden nutzerdefinierte Operatoren in Prozeduren transformiert und in der abstrakten Syntax ein Verweis auf die Prozedur hinterlegt.

Vordefinierte Operatoren (also Operatoren aus dem Paket predefined) können nicht auf die gleiche Weise interpretiert werden, da für sie keine Definition mit SDL-Mitteln angegeben werden kann. Deshalb muss die Berechnung von vordefinierten Operatoren „direkt“ in der Datentypsemantik erfolgen.

Zur Bewertung eines Operatorrufs muss die dynamische Semantik also folgende Fragen beantworten:

- Gegeben sei ein Operatorruf: Ist das der Ruf einer vordefinierten (funktionalen) Operation, oder erfordert er die Abarbeitung einer Prozedur?
- Falls eine Prozedur interpretiert werden soll: Welche?
- Anderenfalls, gegeben die Operationsargumente, wie lautet das Operationsergebnis?

Aus diesen drei Fragen wurde die folgende Operationssignatur abgeleitet:

$functional: PROCEDURE \times VALUE^* \rightarrow BOOLEAN$
$dispatch: PROCEDURE \times VALUE^* \rightarrow IDENTIFIER.$
$compute: PROCEDURE \times VALUE^* \rightarrow VALUEORECEPTION$

$PROCEDURE =_{DEF} STATIC\text{-}OPERATION\text{-}SIGNATURE \cup DYNAMIC\text{-}OPERATION\text{-}SIGNATURE \cup LITERAL\text{-}SIGNATURE$

25. Nutzerdefinierte Operatoren können natürlich auch parameterbehaftete Ausnahmen auslösen. Wie in 7.3.3 dargestellt, werden diese jedoch in Prozeduren transformiert und von der Agentensemantik interpretiert.

Mit der Funktion functional wird ermittelt, ob eine gegebene Operation für eine gegebene Argumentliste funktional ist (also eine vordefinierte Semantik hat), oder durch eine Prozedur definiert wird. Die Funktion dispatch ermittelt in letzterem Fall die zu interpretierende Prozedur und gibt deren Bezeichner zurück. Für vordefinierte Operatoren ermittelt die Funktion compute den Ergebniswert (der unter Umständen eine Ausnahme ist).

Beim Versuch, die Funktion compute zu definieren, zeigte sich, dass diese Schnittstelle unzureichend für die SDL-Semantik ist: auch die Berechnung vordefinierter Operatoren kann unter Umständen Seiteneffekte haben, wie beispielsweise die verändernden Operatoren für Strukturtypen und Felder. Deshalb wurde eine Funktion computeSideEffects in die Schnittstelle aufgenommen, die den neuen Agentenzustand ermittelt:

computeSideEffects: $\mathit{PROCEDURE} \times \mathit{VALUE}^* \times \mathit{STATE} \times \mathit{STATEID} \rightarrow \mathit{STATE}$

8.1.5 Semantische Werte

Mit den soweit definierten Funktionen kann die Agentensemantik die Datentypsemantik einbetten, also alle nötigen Berechnungen ausführen. Tatsächlich reicht das aber zur Definition der Agentensemantik nicht. Wenn beispielsweise das Decision-Konstrukt interpretiert werden soll, so muss nicht nur der Testausdruck ausgewertet und mit den möglichen Antworten verglichen werden. Es muss auch noch ermittelt werden, ob die Antwort wahr oder falsch ist.

Aus diesem Grund wurde eine Reihe von Funktionen definiert, die für einen Wert der Domäne *VALUE* ermittelt, was der zugrunde liegende Wert einer der vordefinierten ASM-Domänen ist:

semvalueBool: SDLBOOLEAN $\rightarrow$ *BOOLEAN*
semvalueInt: SDLInteger $\rightarrow$ *Nat*

Dabei ermittelt die Funktion semvalueBool den zugrunde liegenden Wahrheitswert und die Funktion semvalueInt die zugrunde liegende Zahl.

8.2 Integration von Ausdrücken in die Kompilationsfunktion

Ein SDL-Agent wird in der dynamischen Semantikdefinition durch einen ASM-Agenten repräsentiert. Dieser ASM-Agent führt nach der Initialisierung ein festes ASM-Programm aus, welches aus einer Menge von Elementaraktionen (siehe Abschnitt 5.5) besteht. Diese Elementaraktionen sind vom Typ *PRIMITIVE* und werden durch die Kompilationsfunktion compile zugeordnet:

BEHAVIOUR $=_{\text{def}}$ *PRIMITIVE*-**set**
PRIMITIVE $=_{\text{def}}$ *LABEL* $\times$ *ACTION*
compile: *DEFINITIONAS1* $\rightarrow$ *BEHAVIOUR*

Die Kompilationsfunktion ist als **case**-Ausdruck definiert, der für alle Elemente des Graphen aller Agenten eine Menge von *PRIMITIVE*-Werten ermittelt. Jeder dieser Werte enthält einen Wert aus der Domäne *LABEL*. Über diese Werte wird die Reihenfolge von Aktionen festgelegt: Element jeder Aktion, die eine Folgeaktion besitzt, ist der *LABEL*-Wert dieser Folgeaktion.

Das Ergebnis der Kompilation jedes Transitionsgraphen wird von einem ASM-Agenten abgearbeitet, der für einen *PRIMITIVE*-Wert eine ASM-Aktion ausführt, die unter anderem festlegt, welcher *PRIMITIVE*-Wert im folgenden Schritt ausgeführt wird.

Beispiel 41. In der abstrakten Syntax 1 wird eine **return**-Anweisung, die den Wert einer Prozedur zurückgibt, durch den Knoten *Value-return-node* repräsentiert. In der Kompilationsfunktion wird dieser Knoten auf folgende Weise in *PRIMITIVE*-Werte übersetzt[26]:

26. Die compile-Funktion ist als case-Ausdruck definiert, dessen Alternativen aus Teilfragmenten wie dem hier angegebenen zusammengesetz werden.

```
| v=Value-return-node(expr) =>
    compileExpr(expr, uniqueLabel(v,1)) ∪
      {mk-PRIMITIVE(uniqueLabel(v,1), mk-RETURN(uniqueLabel(expr,1))) }
```

Die Funktion uniqueLabel generiert *LABEL*-Werte, die in Bezug auf den Knoten der abstrakten Syntax und die laufende Nummer (hier 1) eindeutig sind. Ergebnis der Kompilation ist ein *PRIMITIVE*-Wert, der eine *RETURN*-Aktion sowie das Kompilationsergebnis des Ergebnisausdrucks enthält. Zur Interpretation des Syntaxknotens *Value-return-node* werden zunächst die Aktionen zur Berechnung des Ausdrucks durchgeführt. Diese assoziieren das Ergebnis der Berechnung mit einem *LABEL*-Wert. Die Interpretation der *RETURN*-Aktion liest den Ergebniswert und assoziiert ihn mit einem *LABEL*-Wert des Rufers der Prozedur.
Die eigentliche Ausführung der Return-Anweisung erfolgt durch das ASM-Makro EvalReturn:

```
EvalReturn(a: Return) ≡
      if a.s-implicit ∈ VALUELABEL then
         EvalExitProcedure(a.s-implicit )
      else
         EvalExitCompositeState(a.s-implicit)
      endif
```

Die Elementaraktion *RETURN* wird sowohl zum Verlassen von Prozeduren als auch zum Verlassen von verschachtelten Zuständen verwendet. Die Interpretation dieser Aktion testet also, welcher der beiden Fälle vorliegt, und führt für Prozeduren EvalExitProcedure aus. Dieses Makro entfernt die aktuelle Prozedur vom Prozedurstack und speichert den Rückgabewert der Prozedur, so dass im Rufer dieser Wert verwendet werden kann.

Zur Kompilation von Ausdrücken dient die Funktion compileExpr, die neben dem zu übersetzenden Knoten der abstrakten Syntax auch den *LABEL*-Wert der Folgeaktion erhält:

```
compileExpr: DEFINITIONAS1 × LABEL → Behaviour
```

Beispiel 42. Die Übersetzung eines Prozedur- oder Operatorrufs erfolgt durch die **case**-Alternative

```
| v=Value-returning-call-node(*, procedureId, exprList) =>
      if exprList = empty then ∅
      else compileExpr(exprList.last, uniqueLabel(v,1))
         U{ compileExpr(exprList[i],  exprList[i+1]. startLabel) | i ∈ 1.. exprList.length - 1 }
      endif ∪
      (let paramDef = procedureId.idToNodeAS1.s-Procedure-formal-parameter-seq  in
         {mk-PRIMITIVE(uniqueLabel(v,1),
            mk-CALL(procedureId,
               < ( if paramDef[idx] ∈ In-parameter
                 then uniqueLabel(exprList[idx], 1)
                 else exprList[idx]
                 endif )
               | idx in (1..exprList.length )>,
               next)) }
      endlet)
```

Zur Übersetzung des Aufrufs werden zunächst die Ausdrücke der Prozedurargumente übersetzt und dann eine *CALL*-Aktion generiert, die als Argument die zu rufende Prozedur sowie die Liste der *LABEL*-Werte bekommt, mit denen die Argumente assoziiert sind. Die Interpretation dieser Aktion wird zunächst überprüfen, ob es sich um einen vordefinierten Operator handelt, und anderenfalls mit der Interpretation der Prozedur beginnen.

8.3 Objekttypen

Variablen, deren Sorte auf einem Objekttyp beruht, besitzen als Wert Referenzen auf andere Werte, oder sie sind Null-Referenzen (also ohne assoziierten Wert). Die Referenz stellt eine Objektidentität dar. Bei Erzeugung eines Objekts wird die Identität geschaffen und mit einem Wert assoziiert. Diese Assoziation kann durch Modifikation des Objekts geändert werden, seine Identität jedoch bleibt bestehen.

SDL-2000 bietet keine Konstrukte zur Zerstörung von Objekten: Ein einmal geschaffenes Objekt bleibt konzeptionell so lange bestehen wie der Agent, in dem es geschaffen wurde[27]. Damit entsprechen Objektidentitäten einer dynamischen Domäne des ASM-Kalküls, es wäre also naheliegend, bei Erzeugung eines Objekts eine solche Domäne über die **extend**-Aktion zu erweitern.

Leider verletzt diese Modellierung die Forderung, dass die Datentypsemantik eine funktionale Schnittstelle bieten soll, da die Erweiterung einer Domäne den Systemzustand ändert.

Deshalb wurde eine **shared**-Domäne definiert und eine Funktion, die in jedem Systemzustand einen neuen Wert dieser Domäne liefert:

shared domain *ObjectIdentifier*
initially *ObjectIdentifier* = { null }
mkObjectId: → *ObjectIdentifier*

Durch diese Funktion kann in der Datentypsemantik in jedem Interpretationsschritt ein neues Objekt erzeugt werden. Tatsächlich ist das für die Realisierung der SDL-Semantik ausreichend, da in einem Interpretationsschritt höchstens ein Operator aufgerufen wird, und jeder Operator höchstens ein neues Objekt erzeugt.

Auf Basis dieser dynamischen Domäne wurde die Domäne *ObjectValue* definiert, die die Menge aller Objekte eines Agenten repräsentiert.

$Object =_{def} ObjectIdentifier$
$ObjectValue =_{def} Object \times Value$

Der unmittelbare Wert einer Variablen eines Objekttyps ist damit ein Wert der Domäne *Object*. Falls dieser Wert nicht null ist, gibt es im Agentenzustand eine Zuordnung von diesem Wert zu einem Wert der Domäne *Value*.

8.4 Repräsentation von Zuständen

Obwohl die Domäne *State* Teil der Schnittstelle der Datentypsemantik ist, ist ihre Definition es nicht: Die Konstruktion und Verarbeitung von *State*-Werten obliegt ausschließlich der Datentypsemantik.

Wie in Abschnitt 8.1.2 dargestellt, muss die Repräsentation von Zuständen folgende Aspekte berücksichtigen:

- Die Zustände von Agenten stellen eine Hierarchie dar. Innere Agenten und Prozeduren können auf die Variablen der äußeren Agenten zugreifen.
- Die Angabe des Variablennamens allein reicht nicht, um den Variablenwert zu identifizieren. Es ist auch die Kenntnis des Agenten nötig, in dem der Zugriff erfolgt.

Aus diesen Beobachtungen ergab sich die Forderung, dass der Zustand von Agenten stets mit dem äußersten Prozessagenten assoziiert sein muss.

Während der Realisierung des Zustandsmodells wurde beobachtet, dass weitere Aspekte beachtet werden müssen:

27. Implementierungen von SDL-2000 können natürlich den Speicher für das Objekt freigeben, wenn es nicht länger referenziert wird.

- Verschachtelte Agenten und Prozeduren können Referenzen auf die gleichen Objekte besitzen. Die Zuordnung von Objektidentität zu Objektwert muss also ebenfalls „global“ für den äußersten Prozessagenten erfolgen.
- Prozeduren können mit Ein- und Ausgabeparametern definiert werden. Die aktuellen Argumente für Ausgabeparameter müssen Variablen sein. Innerhalb des Prozedurrufs ist dann der Name des formalen Parameters ein Synonym für den Namen der Variable des Rufers: Zuweisungen an den formalen Parameter ändern sofort die Variable des Rufers.

Um die Belegung von Variablen zu modellieren, ist es naheliegend, sie als Paare von Variablenbezeichner und Wert zu betrachten. Da auch der zugreifende Kontext den Wert beeinflusst, müssen sie tatsächlich Tripel sein, bei denen ein Element aus der Domäne *StateId* stammt.

Um jedoch Ausgabeparameter repräsentieren zu können, reicht das Modell, Variablennamen Werte zuzuordnen, nicht: Manche Variablen sind mit einem weiteren Variablennamen assoziiert, und erst dieser ist dann an einen Wert gebunden. Daraus ergibt sich die Definition der Domäne *BoundValue*:

$$\mathit{BoundValue} =_{\text{def}} \mathit{Value} \cup \mathit{Variable\text{-}identifier}$$

Ein Wert aus *BoundValue* definiert den Wert einer Variablen. Berücksichtigt man, dass manche Variablen keinen Wert haben (also „undefined“ sind), so ergibt sich die Definition der Domäne *NamedValue*:

$$\mathit{NamedValue} =_{\text{def}} \mathit{StateId} \times \mathit{Variable\text{-}identifier} \times [\mathit{BoundValue}]$$

Mit einem Element aus *NamedValue* wird für einen Zustand und eine Variable ein Wert festgelegt.

Um die Verschachtelung von Zuständen zu repräsentieren, muss bekannt sein, welcher äußere Zustand Grundlage welchen inneren Zustands ist. Diese Zuordnung wird durch die Domäne SuperState repräsentiert:

$$\mathit{SuperState} =_{\text{def}} \mathit{StateId} \times \mathit{StateId}$$

In dieser Domäne repräsentieren die ersten Komponenten den Basiszustand und die zweiten Komponenten den enthaltenen Zustand. Die Verschachtelungsstruktur der Zustände lässt sich dann durch eine Menge von *SuperState*-Werten darstellen: Will man beispielsweise zu einem Zustand z den Basiszustand ermitteln, muss man aus der Menge von SuperState-Werten denjenigen ermitteln, dessen zweites Element z ist und von diesem Wert die erste Komponente auswählen.

Mit diesen Definitionen ist es nun möglich, die Struktur des Gesamtzustands eines äußersten Agenten definieren:

$$\mathit{State} =_{\text{def}} \mathit{NamedValue}\text{-set} \times \mathit{SuperState}\text{-set} \times \mathit{ObjectValue}\text{-set}$$

Die erste Komponente des Zustands fasst die Variablenbindungen. Die zweite repräsentiert die Hierarchie der Zustände, und die dritte die Objekte des Agenten.

Für die Werte der Domäne State müssen eine Reihe von Konsistenzbedingungen gelten, die durch die Definition nicht automatisch erfüllt sind:

- Für jedes Paar (*StateId*, *Variable-identifier*) darf es höchstens ein *NamedValue*-Tripel geben. Anderenfalls hätte die Variable gleichzeitig mehrere assoziierte Werte. Gleichermaßen darf es für jeden *Object*-Wert höchstens ein *ObjectValue*-Paar geben.
- Falls ein *BoundValue*-Wert o aus der Domäne *Object* stammt, muss es einen *ObjectValue* geben, dessen erste Komponente o ist.
- Für jeden *StateId*-Wert darf es höchstens einen Basiszustand geben. Genau ein StateId-Wert darf keinen Basiszustand haben.
- Jeder *StateId*-Wert, der in einem *NamedValue*-Tripel auftaucht, muss gültig sein, dass

heißt, er muss als Komponente eines *SUPERSTATE*-Paares auftauchen. Wenn es keine *SUPERSTATE*-Paare gibt, müssen alle *STATEID*-Werte gleich sein.

Es ist eine Beweisverpflichtung, dass die Funktionen der Datentypsemantik nur Zustandswerte erzeugen, die diese Bedingungen erfüllen.

8.5 Vordefinierte Datentypen

Zur Definition vordefinierter Datentypen kann auf die Domänen zurückgegriffen werden, die als Basiskonzepte für den ASM-Formalismus definiert wurden (siehe Abschnitt 5.3.2). So ist beispielsweise die Menge der ganzen Zahlen, die Domäne *NAT*, als Teil des Formalismus vordefiniert. Da viele der Operatoren in SDL die „übliche“ mathematische Bedeutung haben, kann die Semantikdefinition dieser Operatoren auf die Operationen des Formalismus zurückgreifen.

Allerdings zeigt sich, dass es eine Reihe wesentlicher Abweichungen gibt:

- Die gleiche Domäne der formalen Semantik muss zur Definition verschiedener Sorten verwendet werden. So besitzen beispielsweise die SDL-Sorten Real, Time und Duration die gleiche algebraische Struktur, die Sorten sind jedoch disjunkt. Da SDL-2000 Mechanismen der Laufzeit-Typidentifikation bietet, muss jeder Wert Informationen über die Sorte besitzen, der er entstammt.
- Manche Operationen haben in der Mathematik keine „übliche“ Bedeutung, und sind demzufolge nicht Teil des Formalismus. Beispielsweise gibt es verschiedene Definitionen einer Ganzzahldivision, falls einer der Operanden negativ ist. Die SDL-Semantik legt sich auf eine dieser Bedeutungen fest, in der formalen Semantik muss diese Bedeutung dann mit Hilfe der vorhandenen Operatoren modelliert werden.

Zur Berechnung des Werts eines vordefinierten Operators bietet die Datentypsemantik die Funktion compute (siehe Abschnitt 8.1.4). Diese Funktion ist wie folgt definiert

```
compute (procedure: PROCEDURE, values: VALUE* ) : VALUEOREXCEPTION =def
        if  intype (procedure, IntegerType.s-Name) then computeInteger(procedure, values)
        elseif  intype (procedure, BooleanType.s-Name) then computeBoolean(procedure, values)
        elseif  intype (procedure, CharacterType.s-Name) then computeChar(procedure, values)
        elseif  intype (procedure, RealType.s-Name) then computeReal(procedure, values)
        elseif  intype (procedure, DurationType.s-Name) then computeDuration(procedure, values)
        elseif  intype (procedure, TimeType.s-Name) then computeTime(procedure, values)
        elseif  intype (procedure, Stringtype.s-Name) then computeString(procedure, values)
        elseif  intype (procedure, Arraytype.s-Name) then computeArray(procedure, values)
        elseif  intype (procedure, Powersettype.s-Name) then computePowerset(procedure, values)
        elseif  intype (procedure, Bagtype.s-Name) then computeBag(procedure, values)
        else
            OutOfRange
        endif
```

Diese Funktion ermittelt zunächst, um welchen Operator es sich bei der Prozedur handelt. Dazu bestimmt die Funktion intype, ob der Operator, der als erstes Argument übergeben wird, aus dem Datentyp stammt, der als zweites Argument übergeben wird. In Abhängigkeit davon wird dann eine Funktion gerufen, die die Operatoren dieses Datentyps unterscheiden kann.

Da diese Funktionen sich ähneln, wird hier lediglich ein Beispiel angegeben, nämlich für die Berechnung der Operatoren des Typs Integer.

8.5.1 Repräsentation von Integer-Werten

Der vordefinierte SDL-Typ Integer beschreibt die Menge ganzer Zahlen, er unterliegt also keiner Wertebeschränkung (anders als die Ganzzahltypen anderer Sprachen, wie etwa C oder Java). Damit ist die vordefinierte ASM-Domäne *NAT* als Basis der Definition von Integer

geeignet. Ursprünglich war diese Domäne definiert als die Menge natürlicher Zahlen. Die SDL-Semantik hätte darauf basierend Integer-Werte als Paare (Vorzeichen, Wert) repräsentieren müssen. Die Definition von *NAT* wurde dann auf alle ganzen Zahlen erweitert, da sich bei der Erweiterung kein methodischer Nachteil ergab und die Definition der Datentypsemantik vereinfacht werden konnte.

Trotzdem kann die Sorte Integer nicht als Synonym von *NAT* definiert werden, da bei Datentypspezialisierung von Integer die spezialisierte Sorte von der Basissorte verschieden ist, aber die gleichen Operationen besitzt. Deshalb wird Integer definiert durch

SDLINTEGER $=_{\text{def}}$ *NAT* $\times$ *Identifier*

Der Bezeichner in einem *SDLINTEGER*-Wert bezeichnet die Datentypdefinition, der der Wert entstammt; üblicherweise also <<package Predefined>> Integer.

Der Zugriff auf den semantischen Wert (welcher auch Teil der Datentypschnittstelle ist), wird dadurch recht einfach; man muss lediglich die *NAT*-Komponente ermitteln.

semvalueInt(*v*:*SDLINTEGER*): *NAT*$=_{\text{def}}$ *v*.**s-***NAT*

8.5.2 Berechnung von Operatoren

Die Operatoren des Typs Integer werden durch die Operation computeInteger definiert.

```
computeInteger (procedure: PROCEDURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(procedure) in
    if procedure ∧ Literal-signature then
        integerLiteral(0,procedure.procName, restype)
    elseif procedure.procName = ì-ì  values.length = 1 then
        mk-SDLINTEGER (0 - values.head.semvalueInt, restype)
    elseif procedure ∈ {"+", "-", "*", "/", "mod", "rem", "<", "<=", ">", ">=" } then
        let val1 = values[1]. semvalueInt in
        let val2 = values[2]. semvalueInt  in
            case procedure.procName of
            |  "+" => mk-SDLINTEGER  (val1+val2, restype)
            | "-"=> mk-SDLINTEGER  (val1 – val2, restype)
            | "*"=> mk-SDLINTEGER  (val1 * val2, restype)
            | "/" =>
                if val2 = 0 then
                    raise(DivisionByZero)
                else
                    mk-SDLINTEGER  (intDiv(val1,val2), restype)
                endif
            | "mod" =>
                if val2 = 0 then
                    raise(DivisionByZero)
                else
                    mk-SDLINTEGER  (intMod(val1,val2), restype)
                endif
            | "rem"=>
                if val2 = 0 then
                    raise(DivisionByZero)
                else
                    mk-SDLINTEGER  (intRem(val1,val2), restype)
                endif
            | "power" =>
                if val2 = 0 then
                    raise(DivisionByZero)
                else
                    mk-SDLINTEGER  (intPower(val1,val2), restype)
```

```
            endif
          | "<" => mk-SDLBOOLEAN (val1 < val2, BooleanType)
          | "<=" => mk-SDLBOOLEAN (val1 ≤ val2, BooleanType)
          | ">" => mk-SDLBOOLEAN (val1 < val2, BooleanType)
          | ">=" => mk-SDLBOOLEAN (val1 ≥ val2, BooleanType)
          endcase
      endlet
      else OutOfRange
  endif
  endlet
```

Falls die SDL-Sorte durch Spezialisierung entstanden ist, ändert sich der Rückgabetyp in Abhängigkeit von den Argumenttypen.

Beispiel 43. Die Addition von Integer-Werten ist durch die Signatur

```
"+"    ( this Integer, this Integer) -> this Integer;
```

definiert. Erfolgt nun eine Spezialisierung von Integer zum Typ Zahl durch die Definition

```
value type Zahl inherits Integer {
}
```

so besitzt die Sorte Zahl ebenfalls einen Operator "+", dessen Ergebnistyp jedoch Zahl ist.

Für diese Operatoren ermittelt die Funktion restype die Sorte, in der der Operator definiert ist – die Ergebnissorte ist (für alle Operatoren von Integer) dann entweder restype oder der Typ Boolean.

Manche der Integer-Operatoren lassen sich nicht auf Operatoren des ASM-Kalküls abbilden, sie werden durch weitere Hilfsfunktionen berechnet.

Beispiel 44. Die Integer-Division von a und b ist in SDL durch Angabe der Axiome definiert.

```
a / 0                          == raise DivisionByZero;
a >= 0 and b > a               == true ==> a / b== 0;
a >= 0 and b <= a and b > 0    == true ==> a / b== 1 + (a-b) / b;
a >= 0 and b < 0               == true ==> a / b== - (a / (- b));
a < 0 and b < 0                == true ==> a / b== (- a) / (- b);
a < 0 and b > 0                == true ==> a / b== - ((- a) / b);
```

Diese aus SDL-92 übernommenen Axiome lassen sich durch folgende Funktion formalisieren (wobei bereits computeInteger den Fall der Division durch 0 behandelt):

```
intDiv(a: NAT, b: NAT):NAT =def
  if a ≥ 0 ∧ b > a then 0
  elseif a ≥ 0 ∧ b ≤ a ∧ b > 0 then 1 + intDiv(a - b, b)
  elseif a ≥ 0 ∧ b < 0 then - intDiv(a, -b)
  elseif a < 0 ∧ b < 0 then intDiv (-a, -b)
  elseif a < 0 ∧ b > 0 then - intDiv (-a, b)
  else 0
  endif
```

8.6 Konstruierte Typen

In der abstrakten Syntax 1 gibt es zwei Arten konstruierter Typen: Strukturtypen und Literaltypen (Choice-Typen wurden durch eine Transformation in Strukturtypen überführt).

Auch für Strukturtypen werden als Teil der statischen Semantikdefinition Transformationen durchgeführt, die die Struktur in eine Reihe von Operationen überführen. Diese Operationen verändern oder ermitteln die Felder eines Strukturwerts.

Da die eigentliche Strukturdefinition in der abstrakten Syntax nicht mehr enthalten ist, muss die Interpretation von Rufen der Struktur-Operationen diese an ihren Namen erkennen: beispielsweise heißt der Operator, der aus einer Struktur den Wert des Felds x ermittelt xExtract (siehe Abschnitt 6.5).

Zur Definition des Verhaltens dieses Operators kann also aus dem Operatornamen auf den Feldnamen geschlossen werden. Diese Eigenschaft wird zur Repräsentation von Strukturwerten verwendet werden. Einzelne Felder werden durch die Domäne *FIELD* repräsentiert:

$FIELD =_{def} Name \times VALUE$

Ein Strukturwert besteht dann aus einer Menge solcher *FIELD*-Werte und sowie dem Bezeichner des Strukturtyps:

$SDLSTRUCTURE =_{def} FIELD\text{-}\mathbf{set} \times Identifier$

Auf der Basis dieser Domänendefinitionen können nun Funktionen definiert werden, die die Operatoren interpretieren. Diese Funktionen ähneln sehr denen, die zur Interpretation vordefinierter Operatoren dienen.

Zur Repräsentation von Literaltypen und ihren Werten in der dynamischen Semantik reicht die Definition der Domäne *SDLLITERAL*

$SDLLITERALS =_{def} Literal\text{-}identifier \times Identifier$

Jedes Literal wird in der dynamischen Semantik durch ein Paar von Bezeichnern repräsentiert, wobei der erste Bezeichner den Namen des Literals angibt und der zweite Bezeichner den Literaltyp. Auch für Literaltypen werden in der dynamischen Semantik Funktionen definiert, die die implizit vorhandenen Operatoren berechnen (siehe Abschnitt 6.5).

8.7 Pid-Typen

Pid-Typen entstehen implizit durch die Definition von Schnittstellen (siehe Abschnitt 6.2). Werte von Pid-Typen bezeichnen SDL-Agenten. In der dynamischen Semantik werden SDL-Agenten durch ASM-Agenten modelliert. Daher liegt es nahe, zur Identifikation von SDL-Agenten Werte der ASM-Domäne *AGENT* zu verwenden:

$SDLAGENT =_{def} AGENT$

Diese Definition reicht allerdings nicht zur Repräsentation aller Aspekte von Pid-Typen:

- Es gibt einen ausgezeichneten Pid-Wert null, der keinen SDL-Agenten bezeichnet.
- Für jeden Pid-Wert muss die Schnittstelle bekannt sein, die der Agent realisiert.

Diese Forderungen führen zu der Definition der Domäne *PID* durch Vereinigung folgender Teildomänen:

$NULLPID =_{def} \{ nullPid \}$
$VALIDPID =_{def} SDLAGENT \times [Interface\text{-}definition]$
$PID =_{def} VALIDPID \cup NULLPID$

Für Pid-Werte sind keine SDL-Operationen definiert, so dass sich die Semantikdefinition für Pid-Werte im Wesentlichen auf diese Domänendefinitionen beschränkt. Zusätzlich wirkt sich die Repräsentation von Pid-Werten auch auf die Interpretation von Zuweisungsversuchen (*assignment attempt*, siehe Abschnitt 3.4) aus: Bei Zuweisung eines Pid-Werts p an eine Variable, deren Typ kein Basistyp der *INTERFACE-DEFINITION*-Komponente von p ist, wird der Wert nullPid an die Variable zugewiesen.

8.8 Polymorphie und dynamische Bindung

Wird für eine Objektvariable eine virtuelle Methode gerufen, so hängt die tatsächlich gerufene Methode vom Typ des Werts ab, den die Variable zum Zeitpunkt des Aufrufs hat. Besitzt die Methode mehrere virtuelle Argumente, so tragen auch deren Typen zur Bestimmung der Methode bei.

Virtuelle Methoden werden in der abstrakten Syntax 1 durch Prozeduren definiert, deren Interpretation außerhalb der Datentypsemantik erfolgt. Aufgabe der Datentypsemantik ist also lediglich die Bestimmung der Prozedur, die interpretiert werden soll. Dazu wurde die Funktion dispatch definiert:

```
dispatch(procedure:PROCEDURE, values:VALUE*): Identifier =def
    if procedure ∈ Static-operation-signature then
        procedure.s-Identifier
    elseif ¬ allVirtualArgsSet(procedure.s-Formal-argument-seq, values) then
        raise(InvalidReference)
    else
        let c = allDynamicCandidates(procedure) in
        let c1 = matchingCandidates(c, values) in
                bestMatch(c1)
        endlet
    endif
```

Diese Funktion unterscheidet drei Fälle:

1. Falls die Operation nicht virtuell ist, so enthält der Knoten der abstrakten Syntax direkt den Bezeichner der zu interpretierenden Prozedur.
2. Anderenfalls wird überprüft, ob alle Argumente von null verschieden sind. Ist ein Argument null, so wird die Ausnahme InvalidReference ausgelöst.
3. Falls alle Argumente angegeben wurden, werden zunächst alle Kandidatendefinitionen der virtuellen Methode ermittelt (also alle Redefinitionen). Unter diesen werden diejenigen ausgewählt, deren Parametertypen nicht spezieller als die dynamischen Argumenttypen sind. Von diesen Signaturen wird schließlich die ausgewählt, die spezieller als alle anderen Signaturen ist. Die statische Semantik hat dazu sichergestellt, dass das stets genau eine Operationssignatur ist. Die Funktion bestMatch gibt für diese dann den Bezeichner der Prozedurdefinition zurück.

8.9 Syntypes

Eine Syntype-Definition impliziert einen Teilbereichstest, der unter anderem bei Zuweisungen durchgeführt werden muss (siehe Abschnitt 6.9). Dazu wird in der Datentypsemantik die Funktion rangeCheck definiert, die für einen Wert bestimmt, ob er im angegebenen Teilbereich liegt:

```
rangeCheck(syntype: Syntype-definition, value: VALUE): BOOLEAN =def
    ∃ cond ∈ syntype.s-Range-condition.s-Condition-item-set:
        conditionItemCheck(cond, value, syntype.s-Parent-sort-identifier)
```

Eine Syntype-Definition kann mehrere alternative Teilbereiche enthalten. Die Funktion rangeCheck liefert den Wert „wahr“, wenn es eine Bedingung gibt, für die die Funktion conditionItemCheck wahr ist:

```
conditionItemCheck(item: Condition-item, value: VALUE , type: Identifier): BOOLEAN =def
    if item ∈ Closed-range then
        conditionItemCheck(item.s-Open-range, value, type) ∧
        conditionItemCheck(item.s2-Open-range, value, type)
    else
        semvalueBool(compute(item.s-Operation-identifier, < item.s-Constant-expression, value>))
```

endif

Dieser Test wiederum ist wahr, wenn

- der Bereich ein abgeschlossenes Intervall ist und der Wert zwischen der unteren und oberen Schranke liegt, oder
- der Bereich ein offenes Intervall ist, und der Vergleich zwischen dem Wert und der Intervallgrenze bezüglich des Vergleichsoperators wahr ist.

Diese Funktion rangeCheck muss überall dort aufgerufen werden, wo laut Sprachdefinition der Teilbereichstest erfolgt.

Diese Definition zeigt ein bisher ungelöstes Problem der Datentypsemantik: Falls der Vergleichsoperator des Teilbereichs kein vordefinierter Operator ist (sondern etwa ein nutzerdefinierter "<"-Operator), so muss die Ermittlung des Teilbereichs durch Interpretation einer Prozedur erfolgen. Die Funktion rangeCheck lässt sich in diesem Fall nicht wie oben angegeben definieren, da die Interpretation von Prozeduren mehrere Schritte eines ASM-Agenten verlangt. Diese Abarbeitung steht im Konflikt zur Forderung, dass die Datentypsemantik über eine funktionale Schnittstelle verfügt. Tatsächlich muss die Interpretation des Teilbereichstests bereits in der Kompilationsfunktion vorgesehen und dann vom ASM-Agenten durchgeführt werden.

8.10 Fazit

Hauptziel des Entwurfs der dynamischen Datentypsemantik war (neben der korrekten Abbildung der informalen Semantik) die konsistente Verwendung des ASM-Kalküls unter Einhaltung einer funktionalen Schnittstelle. Es zeigte sich, dass die Definition der dynamischen Datentypsemantik vor allem zwei Schwerpunkte hat: Die Definition des Zustands eines SDL-Agenten, sowie die Definition der vordefinierten Operationen.

Der Zustandsbegriff nimmt in der Formalisierung zum einen deshalb so großen Raum ein, weil zahlreiche unterschiedliche Abhängigkeiten zwischen verschiedenen Sichtbarkeitsbereichen bestehen, nämlich durch Verschachtelung von Prozessagenten, Zustandstypen und Prozeduren, durch die Referenzsemantik und durch Ein/Ausgabeparameter von Prozeduren. Zum anderen wurde die Definition des Zustandsbegriffs durch die Forderung nach einer funktionalen Schnittstelle verkompliziert.

Der Umfang der Definition vordefinierter Operationen ergibt sich aus ihrer Zahl: Für SDL-2000-Programme sind mehr Datentypen und Operationen vordefiniert als für SDL-92; die axiomatischen Definitionen der Operatoren mussten in algorithmische umgeformt werden.

Auch in der Definition der dynamischen Semantik wurden zahlreiche Probleme mit der SDL-Semantik aufgedeckt, insbesondere in der Integration mit der Semantik von SDL-Agenten; diese Probleme führten dann zu einer Änderung der funktionalen Schnittstelle.

9 Der Compiler SDLC

Um SDL-Anwendern den Zugang zur formalen Semantikdefinition von SDL-2000 zu erleichtern, ist es wichtig, dass die Semantikdefinition „tatsächlich“ ausführbar ist. Dazu reicht es nicht aus, Kalküle zu verwenden, die die Abarbeitung eines SDL-Systems auf der Basis der Sprachdefinition ermöglichen (und auf diese Weise zu einer theoretisch ausführbaren Semantikdefinition zu gelangen), sondern es müssen auch Werkzeuge geschaffen oder existierende Werkzeuge verwendet werden, die letztlich die Interpretation des Systems auf einem Computer durchführen.

Da die formale Sprachdefinition auf dem ASM-Kalkül beruht, also eine Menge von ASM-Programmen darstellt, wäre es theoretisch denkbar, die Sprachdefinition einem ASM-Werkzeug zu übergeben und zu erwarten, dass diese ein ebenfalls zu übergebendes SDL-System ausführt. Praktisch funktioniert diese Umsetzung aus folgenden Gründen nicht:

- Die formale Sprachdefinition basiert auf der abstrakten Syntax 0. Die Umsetzung von konkreter zu abstrakter Syntax ist im ASM-Kalkül nicht formalisiert. Dieser Kalkül eignet sich auch nicht zur Definition von Grammatiken und von Parsern für Grammatiken.
- Die formale Sprachdefinition liegt als Microsoft-Word-Dokument vor und verwendet Notationen, die eigens für SDL-2000 definiert wurden (in Anlehnung an verbreitete Notationen). Die Notationen von ASM-Werkzeugen weichen von den gewählten Notationen ab.
- Es gibt zur Umsetzung der statischen Semantikdefinition eines ASM-Programms Alternativen, die eine bessere Abarbeitungsgeschwindigkeit und Nachvollziehbarkeit der Berechnungen erlauben, ohne die Semantik zu verändern.

Deshalb wurde für die Umsetzung der statischen Semantik Abstand von dem Versuch genommen, die Semantikdefinition direkt von einem ASM-Interpreter ausführen zu lassen – es war nämlich zu erwarten, dass ein solcher Ansatz einen Speicherbedarf verursacht hätte, der die Kapazität der zur Verfügung stehenden Hardware übersteigt: Die Auswahl eines Transformationsschritts verlangt vom ASM-Interpreter, parallel alle Muster für alle Positionen im Baum der abstrakten Syntax zu überprüfen, und dann nicht-deterministisch einen gültigen Transformationsschritt auszuwählen. Ein auf abstrakte Syntaxbäume spezialisiertes Werkzeug kann von vornherein die Menge möglicher Transformationsregeln reduzieren.

Statt dessen wurden „klassische“ Compilerbautechniken [ASU86] eingesetzt, um die statische Semantikdefinition (einschließlich der Kompilationsfunktion) auszuführen. Letztlich entstand ein SDL-Compiler, der als Eingabe ein SDL-Programm erhält und nach weiteren Kompilationsschritten (ASM, C#) ein ausführbares Programm generiert.

Dieser Compiler selbst wird aus der Sprachdefinition generiert, wiederum mit Hilfe eines Compilers. Dieser Compiler wird im folgenden als „SDLC“ bezeichnet.

Der Compiler SDLC ist unter Zuhilfenahme von verschiedenen Programmiersprachen geschrieben worden, die selbst wiederum durch Compiler verarbeitet werden. Diese Zusamenhänge sind in Abbildung 13 verdeutlicht.

In diesem Kapitel werden zunächst die verwendeten Werkzeuge vorgestellt, dann ihre Integration in die Werkzeugkette sowie Probleme und Lösungen, die sich bei der Verwendung der Werkzeuge ergaben. Schließlich wird der Nutzen aus dem Einsatz der Werkzeuge anhand einer Fehlerstatistik quantifiziert.

9.1 Verwendete Werkzeuge

Zur Entwicklung von SDLC kam eine Reihe von Werkzeugen zum Einsatz, die im folgenden kurz vorgestellt werden sollen.

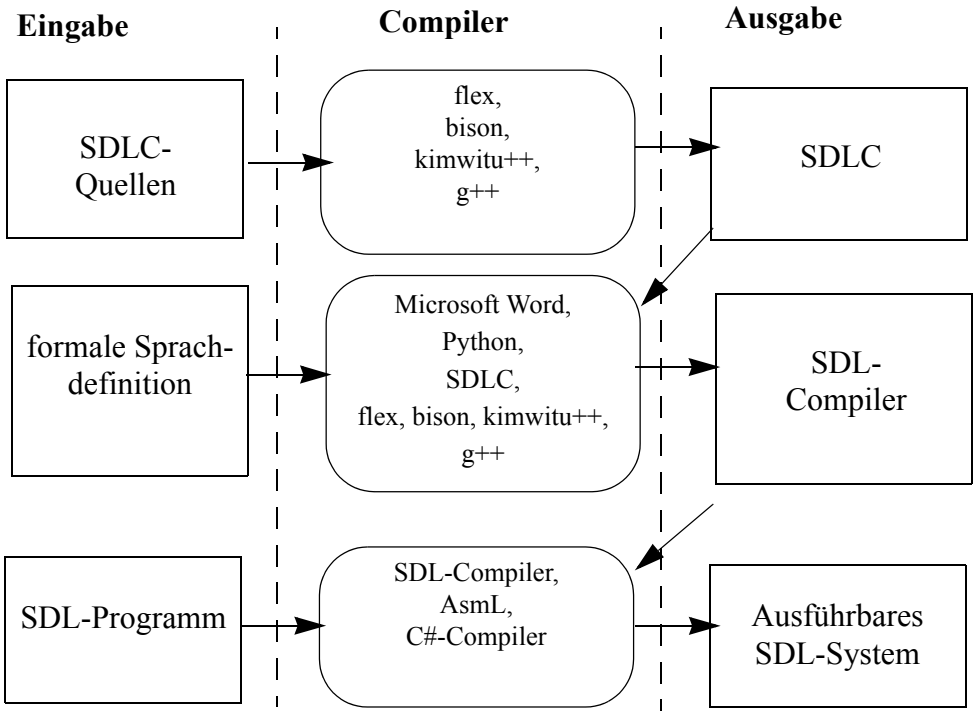

Abbildung 13: Compiler in der Werkzeugkette für SDLC

9.1.1 Flex

Flex ist ein Generator für schnelle lexikalische Analysatoren (*scanner*) [FSF98]. Seine Eingabe ist eine Datei im Format von lex [LMB92], bei der also zum einen eine Menge von regulären Ausdrücken angegeben wird, zum anderen für jeden dieser regulären Ausdrücke eine Aktion.

Nach Übersetzung dieser Datei durch Flex entsteht eine C-Funktion yylex, die die Eingabe eines Compilers in lexikalische Einheiten zerlegt. Jeder Aufruf liefert dabei eine lexikalische Einheit zurück. Um welche lexikalische Einheit es sich dabei handelt, wird von den Aktionen der Flex-Eingabe bestimmt.

Für SDLC wurde die Version 2.5.4 verwendet.

9.1.2 Bison

Bison ist ein Compilergenerator, der aus einer Grammatik in BNF einen LALR-Parser generiert [FSF02a]. Die Eingabesyntax von Bison lehnt sich stark an YACC [LMB92] an.

Jede Regel der Grammatik kann mit einer Aktion annotiert werden. Erkennt der Parser in seiner Eingabe die Symbolfolge der rechten Seite einer Regel, so führt er die dort angegebene Aktion aus, und reduziert die Symbolfolge zu dem Symbol auf der linken Seite der Regel.

Bison generiert aus seiner Eingabe eine Funktion yyparse, die ihrerseits yylex aufruft, um die lexikalischen Einheiten als Terminalsymbol der Grammatik zu verarbeiten. Die Funktion yyparse kehrt erst zurück, wenn die Eingabe vollständig verarbeitet wurde.

Für SDLC wurde die Version 1.35 verwendet.

9.1.3 Kimwitu++

Kimwitu++ ist ein Werkzeug zur Verarbeitung von Termen [Pie02]. Seine Eingabe lehnt sich stark an Kimwitu [EB93] an, jedoch werden semantische Aktionen in Kimwitu++ als C++-Text verstanden, nicht (wie in Kimwitu) als C-Text.

Die Eingabe von Kimwitu++ besteht aus Dateien, in denen folgende Konstrukte verwendet werden können:

- Termdefinitionen
  Mit Hilfe von Termdefinitionen wird eine abstrakte Syntax definiert, die aus *Operatoren* und Phyla besteht. Ein Operator ist ein Knoten der abstrakten Syntax, der eine feste Zahl typisierter Kindknoten besitzt. Ein *Phylum* ist ein Knotentyp der abstrakten Syntax, der als Vereinigung einer Menge von Operatoren definiert wird. Die *Operatoren* heißen Ausprägungen des Phylums.
- Rewrite-Regeln
  Zur Transformation eines Baums der abstrakten Syntax in einen anderen können Rewrite-Regeln formuliert werden. Sie bestehen aus einem Muster und einem Ausdruck. Trifft das Muster auf einen Teil des Baums zu, so wird dieser Teil durch den Ausdruck ersetzt. Rewrite-Regeln können mittels *Rewrite-Views* klassifiziert werden: Die Abarbeitung von Rewrite-Regeln erfolgt stets unter Angabe eines Rewrite-Views. Betrachtet werden dann nur Regeln, in denen dieser Rewrite-View angegeben ist.
- Unparse-Regeln
  Zur Synthese in Compilern können die Unparse-Regeln von Kimwitu++ verwendet werden. Sie bestehen aus einem Muster und einer Aktion. Der Baum wird, beginnend bei der Wurzel, traversiert. Trifft ein Muster auf den gerade betrachteten Knoten zu, so werden die Aktionen ausgeführt. Diese können die Bearbeitung von Kindknoten durchführen. Trifft keine Regel zu, so werden automatisch die Kindknoten traversiert. Durch *Unparse-Views* können die Regeln gruppiert werden. Für einen Unparse-Vorgang kommen nur die Regeln des aktuellen Unparse-Views in Betracht.
- Funktionen und Klassen
  Neben den Kimwitu-spezifischen Konstrukten können auch C++-Fragmente wie Funktions- und Klassendefinitionen Teil der Kimwitu-Eingabe sein. Diese werden in die Ausgabe übernommen.

Kimwitu++ erzeugt aus seiner Eingabe eine Reihe von C++-Dateien als Ausgabe.

Verschiedene Kimwitu++-Versionen wurden für SDLC verwendet, unter anderem 2.3.3.

9.1.4 g++

g++ ist der C++ Compiler aus der Compilersammlung GCC [FSF02b]. Der Compiler implementiert weitestgehend die in [ISO14882] definierte Sprache. Für SDLC wurden verschiedene Versionen verwendet, unter anderem 3.1.

9.1.5 AsmL.NET

AsmL ist eine ASM-Implementierung von Microsoft Research [Mic02]. In der verwendeten Version 2.0 basiert sie auf Microsoft .NET und generiert aus der ASM-Eingabe C#-Programme [Arc01]. Diese werden von VisualStudio.NET in Programme der .NET-Plattform übersetzt.

9.1.6 Python und PyXML

Python ist eine objekt-orientierte Skriptsprache [Pyt02]. Auf deren Basis bildet das Paket PyXML eine Bibliothek zur Verarbeitung von XML-Dateien. Der dort enthaltene Parser sgmlop kann auch HTML-Dateien verarbeiten. Verwendet wurde die Version 2.2 von Python und die Version 0.8.1 von PyXML [vLF00].

9.1.7 Weitere Werkzeuge

Für die Entwicklung von SDLC kamen eine Reihe weiterer Werkzeuge zum Einsatz, die nicht spezifisch für SDLC sind, unter anderem:

- Microsoft Word 2000,
- GNU make 3.79.1 zur Steuerung der Übersetzungsabläufe,
- cvs 1.10 zur Revisionsverwaltung der Quelltextdateien und
- GNU sed 4.0.1 für einfache Filterungsaufgaben.

9.2 Extraktion der formalen Semantik aus Microsoft Word

Wie in Abschnitt 4.1 geschildert, ist die formale Semantik von SDL mit dem Werkzeug Microsoft Word erstellt worden. Das von diesem Werkzeug erzeugte Dateiformat ist für die Weiterverarbeitung durch Werkzeuge Dritter nicht direkt verwendbar, da es vom Hersteller nicht dokumentiert ist und vor allem zur Erstellung von Textdokumenten, nicht aber für den Compilerbau vorgesehen ist.

Aus diesem Grund muss die Microsoft-Word-Datei in ein anderes Format konvertiert werden. In Anbetracht an die in folgenden Verarbeitungsschritten eingesetzten Werkzeuge sollte dieses Format eine reine ASCII-Text-Datei sein. Diese ASCII-Datei muss die gleiche semantische Ausdruckskraft besitzen wie die Formeln, die im Word-Dokument erscheinen.

Um die Verarbeitung vollständig automatisch durchführen zu können, wurden für die Formeln im Word-Dokument eine Reihe von Formatierungsvorschriften festgelegt, die dann in der weiteren Verarbeitung verwendet werden können. Tabelle 5 zeigt den Zusammenhang zwischen der Formatierung in Microsoft Word und der daraus zu extrahierenden ASCII-Version.

Tabelle 5: Umsetzung von Word-Formatierung in ASCII

Word-Formatierung	Beispiel	Verarbeitungs-vorschrift	Beispiel
Zeichenformat ASM-Name	*expr*	Präfix a-	a-expr
Zeichenformat Domainname	*BOOLEAN*	Präfix d-	d-Boolean
Zeichenformat Functionname	*assign*	Präfix f-	f-assign
Zeichenformat AbstractSyntaxName	*Variable-identifier*	Präfix d-	d-Variable-identifier
Zeichenformat SDL-Keyword	**process type**	Präfix kw-	kw-process kw-type
Zeichenformat MacroName	DELIVERSIGNALS	Präfix r-	r-DeliverSignals
Zeichenformat ModuleName	AGENT-SET-PROGRAM	Präfix p-	p-Agent-Set-Program

Zusätzlich müssen Sonderzeichen aus der Schriftart Symbol in eine Textform übertragen werden. Die Liste der Sonderzeichen ist in Tabelle 6 angegeben.

Tabelle 6: Sonderzeichen

Zeichen	Textform	Zeichen	Textform
∀	\forall	⊃	\supset
∃	\exists	⊇	\supseteq
≤	\leq	⊄	\notsubset
→	\rightarrow	⊂	\subset
≥	\geq	⊆	\subseteq
×	\times	∈	\in
≠	\neq	∉	\notin
≡	\equiv	¬	\not
∅	\emptyset	∧	\land
∩	\intersection	∨	\lor
∩	\concat	⇔	\iff
∪	\union	⇒	\implies

Zur Generierung der Textrepräsentation aller Formeln wurde zunächst ein Satz von Visual-Basic-Funktionen entwickelt (siehe [Pri99]). Diese Funktionen können auf die interne Repräsentation des Word-Dokuments zugreifen und Informationen über das Format eines Zeichens oder eines Absatzes erhalten. Um die Text-Version zu erhalten, haben die Funktionen Teile des Originaldokuments in ein neues Dokument kopiert und dabei die nötigen Konvertierungen vorgenommen. Am Ende wurde dieses Dokument in eine Textdatei gespeichert.

Leider sind Feinheiten der internen Repräsentation von Word (beispielsweise die Verwendung der Schriftart Symbol) nicht genau dokumentiert, und es zeigt sich, dass das Visual-Basic-Programm bei Wechsel von Microsoft Word 97 auf Microsoft Word 2000 nicht mehr genau so wie zuvor funktionierte. Die Korrektur eines solchen Programms erweist sich als extrem schwierig, weil es sich bei Abarbeitung unter einem Debugger nicht mehr genau so verhält wie ohne Debugger, und die Größe der Dateien eine Analyse des laufenden Visual-Basic-Programms erschwerte.

Aus diesem Grund hat der Autor dieser Arbeit eine alternative Technologie entwickelt, bei der die Word-Dokumente zunächst in HTML umgewandelt werden. Diese Umwandlung erfolgt mit der in Word 2000 vorhandenen HTML-Export-Routine. In der exportierten HTML-Datei sind nach wie vor alle relevanten Informationen über die Formatierung enthalten, so dass eine Umwandlung in die Textform lediglich die HTML-Form als Eingabe benötigt.

Der Konverter von HTML in die Textform wurde als Python-Programm realisiert, unter Verwendung der Bibliothek PyXML.

Im Ergebnis der Konvertierung entstehen die Dateien, die in Tabelle 7 dargestellt sind. Die

Tabelle 7: Ausgabedateien der Extraktion

Datei	Inhalt
sem2-as1.txt	Abstrakte Syntax 1, wie sie in der statischen Semantik (F.2) angegeben ist.
sem2-fun.txt	Funktionsdefinitionen aus der statischen Semantik.
sem3-as1.txt	Abstrakte Syntax 1, wie sie in der dynamischen Semantik (F.3) angegeben ist.
sem3-fun.txt	Funktionsdefinitionen aus der dynamischen Semantik
sem-as0.txt	Abstrakte Syntax 0
sem-compile.txt	Kompilationsfunktion
sem-cond0.txt	Bedingungen für die abstrakte Syntax 0
sem-cond1.txt	Bedingungen für die abstrakte Syntax 1
sem-map.txt	Mapping-Funktion
sem-asm.txt	ASM-Programme und Funktionen der dynamischen Semantik
sdl-as1.txt	Abstrakte Syntax 1 (extrahiert aus dem Haupttext von Z.100)
sdl-cs.txt	Konkrete Syntax (extrahiert aus dem Haupttext von Z.100)
sdl-lexic.txt	Lexikalische Regeln (extrahiert aus dem Haupttext von Z.100)

Dateinamen drücken zum einen aus, aus welcher Quelle die extrahierten Daten stammen (informaler SDL-Standard, Annex F.2 oder Annex F.3), zum anderen, welche Art von Syntax in ihnen zu finden ist (konkrete oder abstrakte Syntax, Bedingungen, Transformationen, Funktionen, usw.)

9.3 Syntaxanalyse

Wie in [Pie00] dargestellt, wurde die abstrakte Syntax 0 zunächst über ein Generierungsverfahren aus der konkreten Syntax erzeugt, zusammen mit einem Bison-Parser, der aus einer Eingabe in konkreter Syntax einen Baum der abstrakten Syntax 0 generiert. Wie weiter unten erläutert, ist dieser automatisch Schritt jedoch nur ein einziges Mal ausgeführt worden.

Aus der Definition der abstrakten Syntax 0 wurde wiederum eine Kimwitu++-Grammatikdefinition generiert, die für alle Tupel-Regeln der abstrakten Syntax einen Kimwitu++-Operator erzeugt. Der generierte Parser baut in seinen semantischen Aktionen dann aus der Eingabe entsprechend Knoten dieser abstrakten Syntax auf.

Beispiel 45. Die Regel <operation body> der konkreten Syntax lautet

<operation body> ::= [<on exception>] <start> { <free action> | <exception handler> }*

Die entsprechende Regel der abstrakten Syntax 0 lautet (siehe Abschnitt 6.6)

```
<operation body> ::
        [<on exception>] <start> {<free action> | <exception handler>}*
```

Aus dieser Regel wird folgender Kimwitu++-Operator definiert

```
AS0_operation_body( AS0_rule AS0_rule AS0_rule )
```

In der Kimwitu++-Definition der abstrakten Syntax sind alle Knoten Ausprägungen des Phylums AS0_rule, deshalb haben alle Kindknoten von AS0_operation_body den Typ AS0_rule. Aus Kenntnis der konkreten und abstrakten Syntax wurde nun folgendes Bison-Fragment generiert:

```
operation_body:
    on_exception start free_action_v_exception_handler_any
    { $$=AS0_operation_body( $1, $2, $3 ); }
  | /* empty */ start free_action_v_exception_handler_any
    { $$=AS0_operation_body( AS0_UNDEF(), $1, $2 ); }
  ;
```

Das Hilfssymbol free_action_v_exception_handler_any fasst dabei das dritte Element der Produktion <operation body>; die Optionalität von <on exception> wird in zwei Alternativen übersetzt.

Nachdem diese Generierung vollzogen war, wurde die so entstandene abstrakte Syntax Teil der formalen Semantikdefinition. Darauf basierend wurden im folgenden die Transformationsregeln und die Bedingungen der konkreten Syntax formuliert; die abstrakte Syntax 0 lässt sich nun nicht mehr automatisch generieren, da bei Änderungen die schon vorhandenen Formeln angepasst werden müssen.

In Folge wurden in der weiteren Entwicklung der formalen Definition sowohl die konkrete als auch die abstrakte Syntax manuell verändert. Leider ist es damit nicht mehr möglich, den Parser automatisch aus den Grammatikdefinitionen abzuleiten.

Aus diesem Grund wurde im Rahmen dieser Arbeit der initial generierte Bison-Parser als Ausgangspunkt für die Entwicklung eines Parsers genommen und an die aktuelle Eingabe- und Ausgabesyntax angepasst. Dabei mussten folgende Änderungen gemacht werden:

- Sofern die aktuelle Fassung der abstrakte Syntax 0 von der automatisch generierten Version der Grammatik abweicht, meldet der C++-Compiler bei Übersetzung der Aktionen des Bison-Parsers Fehler. Diese wurden durch Anpassung an die aktuelle abstrakte Syntax korrigiert.
- Bison meldet mehrere Tausend Konflikte in der generierten Grammatik, die dazu führen, dass sich gültige Eingabedateien nicht mehr parsen lassen. Diese Fehler wurden zum einen dadurch behoben, dass die Grammatik semantikerhaltend umformuliert wurde, zum anderen dadurch, dass zeitweilig aus dem Parser Alternativen entfernt wurden, die für die zu untersuchenden Beispiele nicht relevant sind. Damit gibt es also SDL-Programme, die der Sprachdefinition folgen, vom Parser jedoch abgelehnt werden. Der umgekehrt Fall ist nicht möglich.

Neben dem handgeschriebenen Parser werden als Teil der Syntaxverarbeitung auch eine Reihe von Dateien automatisch generiert:

- Aus den Definitionen der abstrakten Grammatiken (sem-as0.txt, sem2-as1.txt und sem3-as1.txt) werden Kimwitu++-Operatordefinitionen generiert wie oben beschrieben, von denen im Parser und in den Transformationsmodellen Gebrauch gemacht wird.
- Die beiden Kopien der abstrakten Syntax 1 (in F.2 und F.3) werden strukturell verglichen.
- Aus der Lexikdefinition (sdl-lexic.txt) wird eine Flex-Datei generiert.

9.4 Statische Semantikanalyse

Aus den Formeln der statischen Semantikdefinition werden durch das Werkzeug Satanic (*SDL to ASM nearly immaculate converter*) [Pie00] eine Reihe von Kimwitu++-Dateien erzeugt, die zusammen genommen einen Übersetzer von der abstrakten Syntax 0 in die abstrakte Syntax 1 bilden.

Die Übersetzung der Textdateien aus Tabelle 7 in Kimwitu++ funktioniert auf folgende Weise:

- Die Bedingungen (sem-cond0.txt und sem-cond1.txt) werden in Unparse-Regeln übertragen.
- Die Transformationen (sem-trans.txt) werden in Kimwitu++-Rewrite-Regeln übersetzt.
- Aus den Funktionsdefinitionen (sem2-fun.txt und sem3-fun.txt) werden C++-Funktionen generiert.
- Aus der Mapping-Funktion und der Kompilationsfunktion (sem-map.txt und sem-compile.txt) werden ebenfalls C++-Funktionen generiert.

Im Rahmen dieser Arbeit zeigte sich, dass die in [Pie00] entwickelten Generatoren zwar für die dort verwendete SDL-Teilsprache RSDL ausreichend waren, für die formale Definition von SDL aber nicht mehr. Exemplarisch werden im folgenden einige der Probleme und ihre Lösung vorgestellt.

9.4.1 Freie Variablen in Quantisierungen und Set-Comprehension-Ausdrücken

Die Formeln des verwendeten ASM-Kalküls erlauben die Verwendung von Quantisierungen (Existenz- und All-Aussagen) sowie von *set comprehensions* (Konstruktion einer Menge aus einer anderen durch Angabe eines charakteristischen Prädikats).

Beispiel 46. In folgender Funktion werden die Variablen possibleResultSet und resultSet durch einen set-comprehension-Ausdruck definiert. In letzterer Initialisierung ist wiederum eine Allquantisierung enthalten.

```
getBindingListSet0(c: CONTEXT0): BINDINGLIST0-set =def
    let nameList = c.nameList0 in
    let possibleBindingListSet = nameList.possibleBindingListSet0 in
    let possibleResultSet = {pbl ∈ possibleBindingListSet: isSatisfyStaticCondition0(pbl, c)} in
    let resultSet = {r ∈ possibleResultSet: ∀r' ∈ possibleResultSet: r ≠ r'   ⇒
                        mismatchNumber0(r, c) ≤ mismatchNumber0(r', c)} in
        if |resultSet| = 1 then resultSet
        elseif |resultSet| = 0 then ∅
        elseif (∀bl1, bl2 ∈ resultSet: i∈ 1..bl1.length: bl1[i].s-ENTITYDEFINITION0.entityKind0 ≠ literal ⇒
                representSameDynamicOpSig0(bl1[i].s-ENTITYDEFINITION0,
                                           bl2[i].s-ENTITYDEFINITION0))
            then resultSet
        else ∅
        endif
    endlet
```

Zur Übersetzung nach C++ werden sowohl die Quantisierungen als auch die Set-Comprehension-Ausdrücke in Funktorklassen übertragen. Die Grundmenge (im Beispiel possibleBindingListSet für die Initialisierung von possibleResultSet) wird als Kimwitu++-Liste repräsentiert. Die Filterbedingung wird in eine Methode operator() einer Klasse generiert. Zur Berechnung eines Set-Comprehension-Ausdrucks wird nun die Methode filter der Kimwitu++-Liste aufgerufen und ihr eine Instanz der Filterklasse übergeben.

Diese Methode operator() der Filterklasse erhält als Argument das jeweils aktuelle Element der Grundmenge (pbl für die Initialisierung von possibleResultSet). Zur Berechnung des Aus-

drucks sind aber unter Umständen weitere Variablen nötig, die aus Sicht des Set-Comprehension-Ausdrucks dann freie Variablen sind (im Beispiel die Variable c). Diese Variablen sind lediglich in der Umgebung des Set-Comprehension-Ausdrucks gebunden. Da C++ aber keine verschachtelten Funktionsdefinitionen erlaubt, muss ein spezieller Übergabemechanismus für die freien Variablen eingeführt werden: sie werden an den Konstruktor der Funktorklasse übergeben.

Zur Ermittlung freier Variablen bildet nun Satanic eine Liste der durch die Quantifizierung/ Set-Comprehension gebundenen Variablen, sowie eine Liste aller verwendeten Variablen. Die Differenzmenge dieser Listen ist die Liste der freien Variablen.

In der ursprünglichen Version von Satanic wurde nicht berücksichtigt, dass diese Ausdrücke auch verschachtelt sein können. So ist die Variable r' in der Initialisierung von resultSet eine gebundene Variable, wenn sie innerhalb der Allquantifizierung auftaucht. Ansonsten wäre sie eine freie Variable. Statt dessen hat Satanic diese Variable aus Sicht des Set-Comprehension-Ausdrucks als frei betrachtet – der generierte C++-Code lies sich dann nicht übersetzen, weil eine C++-Variable rPRIME im Kontext des Ausdrucks nicht definiert war. Zur Korrektur dieses Problems musste also berücksichtigt werden, dass die Liste der freien Variablen sich während der Traversierung des Ausdrucks ändern kann.

Beispiel 47. Aus der Initialisierung von resultSet wird nun folgender Code generiert:

```
v_resultSet = unique(v_possibleResultSet->filter(pred_qboo(v_c, v_possibleResultSet)));
```

Die Klasse pred_qboo ist hierbei die Filterklasse. Sie ist definiert als

```
struct pred_qboo : public unaere_funktion<AS0_rule, bool> {
   AS0_rule v_c;
   AS0_rule v_possibleResultSet;
   pred_qboo(AS0_rule c, AS0_rule possibleResultSet):
            v_c(c), v_possibleResultSet(possibleResultSet){ }
   bool operator()(AS0_rule)const;
};
```

Die Methode operator() schließlich ist definiert als

```
bool pred_qboo::operator()(AS0_rule v_r) const
{
    return AS_eq_nil(v_possibleResultSet->filter(pred_dwno(v_c, v_r)));
}
```

Dabei wurde für die All-Quantifizierung in negierter Form das Prädikat pred_dwno generiert. Die Quantifizierung ist also erfüllt, wenn das Filtern der negierten Bedingung eine leere Liste ergibt.

9.4.2 Typisierung von Funktionen und Variablen

In den Formeln der Semantikdefinition sind viele Konstrukte nicht typisiert. Explizite Typdeklarationen gibt es lediglich für Funktionssignaturen. In vielen Fällen ist diese Typisierung auch ausreichend, da im generierten Kimwitu-Code und dem daraus entstehenden C++-Code die Typisierung sogar schwächer ist als die deklarierten Typen in der formalen Semantik: Alle Knotentypen der abstakten Syntax 0 werden in C++ durch den Typ AS0_rule repräsentiert, alle Knotentypen der abstrakten Syntax 1 durch den Typ AS1_rule.

Wird nun der Code einer Funktion generiert, so müssen in der Deklaration der Funktion in C++ gültige C++-Typen verwendet werden. Der Generator Satanic verwendet hierzu eine Heuristik: Fängt der Typ mit einem Kleiner-Zeichen an, wird der Typ AS0_rule ausgegeben. Fängt er mit einem Großbuchstaben an, wird der Typ AS1_rule ausgegeben.

Diese Heuristik hat sich als unzureichend erwiesen, und wurde deshalb auf die folgende Weise verbessert:

- Ist der Parameter- oder Ergebnistyp *BOOLEAN* oder *NAT*, werden die C++-Typen **bool** oder **int** ausgegeben.
- Ist der Parametertyp eine Vereinigung mehrerer Typen, dann wird die Heuristik auf die Teilmengen der Vereinigung angewendet, die dann im Sinne der Heuristik ein einheitliches Ergebnis liefern müssen.

Auch mit diesen Verbesserungen gab es zahlreiche Typfehler im generierten C++. Diese Fehler resultieren vor allem daraus, dass auch innerhalb einer Funktion Typnamen verwendet werden müssen.

Beispiel 48. In einer **let**-Anweisung muss die gebundene Variable deklariert werden, etwa für

```
let c = allDynamicCandidates(procedure) in
let c1 = matchingCandidates(c, values) in
    bestMatch(c1)
endlet
```

In diesem Ausdruck müssen für die ASM-Variablen c und c1 C++-Variablen deklariert werden. Diese müssen den gleichen Typ haben wie der Rückgabetyp von allDynamicCandidates und matchingCandidates.

Beispiel 49. In einem set-comprehension-Ausdruck muss für die Laufvariable eine C++-Variable deklariert werden. Gegeben sei etwa der Ausdruck

```
{ p | p ∈ Operation-signature:
    p.s-Operation-name = procedure.s-Operation-name }
```

In diesem Ausdruck muss für die Variable p eine C++-Variable deklariert werden, die für jedes Element aus der Domäne *Operation-signature* immer wieder neu belegt wird, um den Testausdruck zu berechnen.

Zur Deklarierung dieser Variablen wurde in [Pie00] eine Heuristik eingesetzt, nach der der Typ solcher Variablen gleich dem zuletzt deklarierten Typ (also in der Regel gleich dem Rückgabetyp der aktuellen Funktion) ist. Es zeigte sich, dass diese Heuristik in vielen Fällen ungeeignet war. Sie wurde deshalb auf die folgende Weise verbessert:

- Wird in einem set-comprehension-Ausdruck über eine Domäne iteriert, so wird der Typ dieser Domäne zur Deklaration der Laufvariable verwendet.
- Für jede Funktion wird ihr Rückgabetyp aufgezeichnet, und zwar durch Generierung von C++-Makros. Für Beispiel 48 würde für die Funktion allDynamicCandidates das Makro

  ```
  #define allDynamicCandidates_AS AS1_rule
  ```

  generiert werden, da die Rückgabewerte dieser Funktion Knoten der abstrakten Syntax 1 sind. Muss nun eine **let**-Variable deklariert werden, deren Wert durch einen Funktionsruf entsteht, so kann diese Variable durch Verwendung des Alias-Namens für den Funktionstyp deklariert werden, für das Beispiel also

  ```
  allDynamicCandidates_AS v_c = all_DynamicCandidates(v_procedure);
  ```

  (Zur Übersetzung von ASM-Variablennamen in C++ wird der Präfix v_ generiert).

Leider zeigte sich, dass auch mit diesen neuen Heuristiken noch nicht alle Typfehler behoben werden können. Verbleibende Probleme entstehen unter anderem durch folgende Konstrukte:

- Manche Funktionen haben einen polymorphen Rückgabetyp. Beispielsweise gibt die Funktion take ein beliebiges Element einer Menge zurück. Dieses Element hat dann den Typ, den alle Elemente der Menge haben. Der Rückgabetyp der Funktion hängt also vom Argu-

menttyp ab.

- Die Ermittlung des Ausdrucks eines Typs erfolgt lediglich für Funktionen. So ist in Beispiel 49 der Typ des Ausdrucks *p*.**s-***Operation-name* AS1_rule, da auf ein Feld der abstrakten Syntax zugegriffen wird. Da es sich aber nicht um einen Funktionsruf handelt, kann auch die Heuristik für Funktionsrufe nicht verwendet werden. Zwar wäre es möglich, in diesem Fall die Heuristik zu erweitern – allerdings gibt es viele weitere Ausdrücke, für die dann ebenfalls Heuristiken gefunden werden müssten.

Zur Vereinfachung des Problems wurde deshalb von Polymorphie und Überladung in C++ Gebrauch gemacht. Es wurde eine Klasse ASUNKNOWN_rule definiert, die polymorph Werte sowohl vom Typ AS0_rule als auch vom Typ AS1_rule aufnehmen kann:

```
struct ASUNKNOWN_rule{
   AS0_rule as0;
   AS1_rule as1;

   ASUNKNOWN_rule():as0(0),as1(0){}
   ASUNKNOWN_rule(AS0_rule r):as0(r),as1(0){}
   ASUNKNOWN_rule(AS1_rule r):as0(0),as1(r){}

   // default copy ctor does the right thing

   operator AS0_rule()const {assert(as0);return as0;}
   operator AS1_rule()const {assert(as1);return as1;}

   ASUNKNOWN_rule* operator->(){return this;}
   ASUNKNOWN_rule filter(filtUc);
   ASUNKNOWN_rule filter(filt0c);
   ASUNKNOWN_rule map(mapUc){abort();}
   ASUNKNOWN_rule map(map0c){abort();}

   int length();
};
```

Diese Klasse wird überall dort eingesetzt, wo eine Typermittlung nicht möglich ist. Wenngleich damit eine Reihe von Typfehlern beseitigt werden konnten, wurde nun eine Reihe von Funktionsaufrufen nun mehrdeutig: Es gibt in dieser Klasse Konvertierungsoperatoren sowohl für AS0_rule als auch für AS1_rule. Ist eine Funktion für beide abstrakte Syntaxbäume definiert, erklärt der C++-Compiler den Ruf als mehrdeutig.

Zur Lösung dieses Problems müssen zu den vorhandenen überladenen Funktionen weitere hinzugefügt werden.

Beispiel 50. Zur Auswahl eines beliebigen Elements aus einer Menge ist die Funktion take in zwei Varianten definiert:

```
AS0_rule take(AS0_rule);

AS1_rule take(AS1_rule);
```

Wird ein Wert vom Typ ASUNKNOWN_rule an take übergeben, meldet der C++-Compiler eine Mehrdeutigkeit. Durch Einführung der Funktion

```
ASUNKNOWN_rule
take(ASUNKNOWN_rule r)
{
  if(r.as0)
    return take(r.as0);
  else
    return take(r.as1);
}
```

kann diese Mehrdeutigkeit beseitigt werden: Die zusätzliche überladene Funktion kann gerufen werden, ohne einen nutzerdefinierten Konvertierungsoperator zu verwenden, und wird damit vom Compiler bevorzugt. Die Implementierung der Funktion entscheidet dynamisch anhand des Inhalts des ASUNKNOWN_rule-Objekts, welche der take-Versionen gerufen werden soll.

In vielen Fällen muss der Rückgabewert dieser zusätzlichen Funktionen wiederum ASUNKNOWN_rule sein (so wie im Beispiel).

Durch Kombination dieser Techniken ist es möglich, die C++-Typfehler sämtlich zu beseitigen.

9.5 Generierung von AsmL

Zur Abarbeitung eines SDL-Systems im ASM-Interpreter sind verschiedene ASM-Definitionen nötig, die aus verschiedenen Eingaben generiert werden.

- Aus der Definition der Agentenprogramme (sem-asm.txt) wird ein Teil des SDL-Laufzeitsystems (sem-asm.asml) generiert.
- Aus der Definition der abstrakten Syntax 1 (sdl-as1.txt) wird ein weiterer Teil des Laufzeitsystems (sdl-as1.asml) generiert.
- Aus dem Baum der abstrakten Syntax 1 wird ein ASM-Ausdruck generiert, dessen Ausführung eine Kopie des Baums im ASM-Interpreter erzeugt.
- Aus dem Ergebnis der Kompilationsfunktion wird ein ASM-Ausdruck erzeugt, der das Verhalten der SDL-Agenten in Form von Primitiven repräsentiert.

Zusätzlich zu diesen generierten AsmL-Dateien gibt es eine Reihe handgeschriebener Eingabedateien für den AsmL-Compiler, die zusätzliche elementare Typen definieren. Diese Typen sind zwar aus Sicht der formalen Sprachdefinition von SDL vordefiniert, nicht jedoch im verwendeten ASM-Interpreter AsmL.

9.6 Abarbeitung der SDL-Spezifikation

Aus den in Abschnitt 9.5 beschriebenen AsmL-Dateien generiert der AsmL-Compiler eine Reihe von C#-Dateien [Arc01], die dann wiederum vom C#-Compiler in ein Microsoft-.NET-Programm übersetzt werden. Dieses Programm wird von der .NET-Laufzeitumgebung ausgeführt und liefert als Ausgabe eine Ablaufverfolgung des ursprünglichen SDL-Systems.

9.7 Fehlerstatistik

Bei der Überarbeitung der formalen Semantikdefinition durch die in diesem Kapitel beschriebenen Werkzeuge wurden eine Reihe von Fehlern gefunden und behoben. Tabelle 8 gibt eine Übersicht über die Art und Anzahl dieser Fehler.

Tabelle 8: Fehler in der Semantikdefinition

Fehlerart	Fehleranzahl in Teil F.2	Fehleranzahl in Teil F.3
Syntaxfehler	352	189
Typfehler	88	314
Falschschreibungen von Bezeichnern	115	80

Tabelle 8: Fehler in der Semantikdefinition

Fehlerart	Fehleranzahl in Teil F.2	Fehleranzahl in Teil F.3
Fehlende oder überflüssige Parameter	159	34
Kleinere semantische Fehler	57	227
Falsche Verwendung der abstrakten Syntax	144	83
Undefinierte Funktionen		32
Probleme mit der abstrakten Syntax 0	4	
Schwere semantische Fehler	12	3

Die Zahl der Fehler in jeder Fehlerklasse lässt sich in vielen Fällen aus der Art der Formalisierung schließen:

- Syntaxfehler
  Syntaxfehler entstehen bei falscher Verwendung der Formelsprache (Klammerfehler, fehlende Kommata u.ä.). Die Zahl der Fehler steht in Relation zur Größe des jeweiligen Teils: der Teil F.2 ist ungefähr doppelt so groß wie der Teil F.3.
- Typfehler
  Typfehler entstehen bei Verletzung des Typsystems der Werkzeuge. Das Verhältnis der Fehler lässt sich mit den verwendeten Werkzeugen begründen: der AsmL-Compiler führt einen wesentlich strikteren Typtest aus als Kimwitu++ und als der C++-Compiler. Das legt die Vermutung nahe, dass im Teil F.2 weiterhin eine große Zahl von Typfehlern enthalten sind.
- Falschschreibungen von Bezeichnern
  Auch hier lässt sich das Verhältnis der Fehlerzahlen teilweise aus der Größe der jeweiligen Teile begründen. Bei sauberer Verwendung der Werkzeuge, insbesondere der Querverweise in Microsoft Word, hätten diese Fehler überhaupt nicht auftreten können, da jede Verwendung eines Bezeichners ein Verweis auf seine Definition sein müsste. Dass trotzdem Fehler dieser Art aufgetreten sind, liegt zum Teil an der Unhandlichkeit der Eingabe von Querverweisen, zum anderen daran, dass diese Technik sich nicht auf alle Bezeichner anwenden ließ (siehe Abschnitt 4.2.2).
- Kleinere semantische Fehler
  Diese Fehler sind bei der Abarbeitung von SDL-Spezifikationen entdeckt worden; das tatsächlich beobachtete Verhalten weicht vom gewünschten ab. Die entdeckten Fehler in der dynamischen Semantik waren vor allem systematische Fehler in der Abarbeitung von Transitionen: der gleiche Fehler musste an verschiedenen Stellen korrigiert werden.
- Falsche Verwendung der abstrakten Syntax
  Da die abstrakte Syntax 0 nur in Teil F.2 verwendet wird, ist in diesem Teil die Zahl der Fehler dieser Klasse höher.
- Neue Hilfsfunktionen
  Zur Behebung anderer Fehler mussten unter Umständen neue Funktionen eingeführt werden. Die Mehrzahl dieser Fehler lag in der dynamischen Datentypsemantik, die Funktionen verwendet hat, ohne eine Definition für sie anzugeben.
- Probleme mit der abstrakten Syntax 0
  Diese Fehler können naturgemäß nur in Teil F.2 auftreten. Ihre Lösung steht noch aus. Zu diesen Fehlern gehört beispielsweise das in Beispiel 9 beschriebene Problem.
- Schwere semantische Fehler

Zu diesen Fehlern gehört die unvollständige oder falsche Formalisierung. Bisher sind mehr Fehler dieser Art in Teil F.2 aufgefallen, weil die Beseitigung dieser Fehler die Voraussetzung dafür ist, dass Spezifikationen überhaupt von Teil F.3 verarbeitet werden können. Beispiele für behobene Fehler sind All-Quantifikationen über unendliche Domänen oder die unvollständige Realisierung des Modells für entfernte Prozeduren.

9.8 Fazit

Zur Umsetzung der formalen SDL-Semantik in ein Computerprogramm wurden zahlreiche verschiedene Werkzeuge eingesetzt. Nur durch den Einsatz dieser spezialisierten Werkzeuge konnten die Aspekte der formalen Semantik adäquat und mit vertretbarem Aufwand auf ein Programm abgebildet werden. Leider führt die Zahl dieser Übersetzungsschritte dazu, dass der gesamte Übersetzungsprozess schwer verständlich wird und zur Pflege der Werkzeuge eine beachtliche Einarbeitungszeit nötig ist.

Die Entwicklung der Werkzeuge erfolgte im Wechselspiel mit der Definition der formalen Semantik: Sofern die Werkzeuge korrekt arbeiteten, zeigten sie oft Fehler in der formalen Definition an. Wurden diese Fehler behoben, mussten die Werkzeuge weitere Bearbeitungsschritte ausführen, die dann Fehler in den Werkzeugen selbst aufdeckten.

Trotz dieser Probleme gibt es keine Alternative zum gewählten Entwicklungsweg: Ohne Werkzeugunterstützung wären die Fehler in der formalen Semantikdefinition nie entdeckt worden. Alternative Strategien bestanden lediglich punktuell: Beispielsweise ist es denkbar, dass die Wahl eines anderen Parsergenerators einige der Einschränkungen der konkreten Syntax nicht erforderlich gewesen wären. Da keiner der Versuche, gewisse Werkzeuge durch andere auszutauschen, zu sichtbaren Erfolgen führte, wurde auch keine dieser Alternativen weiter verfolgt.

10 Zusammenfassung, Vergleich und Ausblick

Die objektorientierten Datentypkonzepte von SDL-2000 stellen eine wesentliche Weiterentwicklung gegenüber früheren Sprachversionen dar. Mit dieser Arbeit erhalten sie eine formale Semantik, mit der die Interpretation der Datentypen eines SDL-Systems unzweideutig wird.

Die formale Semantikdefinition von SDL beruht auf dem Kalkül der Abstract State Machines (ASM). Mit diesem Formalismus werden allerdings vorwiegend die aktiven Komponenten eines SDL-Systems, also die SDL-Agenten beschrieben. Zur Integration der passiven Datentypen stellt die Arbeit eine funktionale Schnittstelle vor, deren Funktionen das Verhalten der Datentypen seiteneffektfrei beschreiben. Die Funktionen der Schnittstelle werden in die ASM-Programmdefinition integriert. So ist es möglich, die Datentypsemantik separat von der Semantik von Agenten zu betrachten und weiterzuentwickeln.

Ursprünglich wurde die Datentypschnittstelle auch mit dem Ziel eingeführt, den Datentypteil durch einen anden (etwa den von C++ oder Java) ersetzen zu können. Diese Arbeit zeigt aber, dass eine solche Ersetzung unter Beibehaltung der Schnittstelle nahezu unmöglich sein wird: Zur korrekten Abbildung aller Aspekte der Datentypsemantik wurde die Schnittstelle mehrfach geändert, und weist nun Funktionen auf, die eigentlich nur für die SDL-Datentypsemantik sinnvoll sind. Damit erweisen sich die in [Sch02] vorgestellten Konzepte als einzig praktikable Strategie zur Integration anderer Datentypsysteme in SDL.

Aufgrund des Umfangs der formalen Semantikdefinition lässt sich diese nicht leicht auf einmal erfassen. Um die Semantikdefinition trotzdem verständlich zu machen, ist sie modular gestaltet: Für jeden Aspekt von SDL (wie etwa für jeden vordefinierten Datentyp) wird ein separater Satz von Formeln definiert.

Leider ist es immer noch nicht einfach, den Zusammenhang zwischen den verschiedenen Teilen der Semantikdefinition zu verstehen. Hierbei helfen Werkzeuge: Wenngleich die verwendeten Kalküle theoretisch eine abarbeitbare Semantikdefinition erlauben, wird diese erst durch Werkzeuge auch praktisch abarbeitbar.

Mit Hilfe von Werkzeugen wurden zahlreiche Fehler in der formalen Sprachdefinition entdeck, indem sie beispielsweise die inkorrekte Verwendungen von Funktionen feststellen oder Diagnoseausgaben bei der Abarbeitung einer SDL-Spezifikation liefern.

Neben dem Einsatz von Werkzeugen zur Entwicklung der formalen Sprachdefinition sind die Werkzeuge insbesondere für ihre Verwendung in der Praxis erforderlich: SDL-Anwender müssen sich nicht detailliert mit den Kalkülen und der Strukturierung der Semantikdefinition auseinandersetzen. Sie können formale Semantik als *black box* betrachten, an sie eine SDL-Spezifikation übergeben und die Abarbeitungsergebnisse studieren.

Obwohl bisher diese Werkzeuge nicht tatsächlich von Praktikern verwendet werden, konnte doch an ausgewählten Beispielen demonstriert werden, wie beginnend mit der textuellen SDL-Spezifikation vollautomatisch eine ausführbare Version dieser Spezifikation generiert und diese dann ausgeführt wird.

10.1 Ergebnisse der Arbeit

Diese Arbeit beschreibt die Datentypsemantik von SDL-2000, deren Formalisierung, und die Umsetzung der Formalisierung in Werkzeuge (siehe Abschnitt 1.1). Sie demonstriert, dass es für eine reale Computersprache möglich ist, eine formale Semantik zu definieren, die alle Aspekte der Sprache abdeckt (beginnend bei der syntaktischen Analyse, über die statische Semantik, bis zur dynamischen Semantik). Die Arbeit demonstriert auch, dass für eine reale Sprache die Entwicklung der formalen Semantikdefinition allein auf dem Papier fruchtlos bleiben muss, da die Semantikdefinition dadurch sehr wahrscheinlich mit Fehlern behaftet ist, und Anwendern der Sprache der Zugang erschwert wird.

Die Arbeit vermittelt Einsichten in die Funktionsweise formaler Methoden: Obwohl es mit einer formalen Definition der Sprache theoretisch möglich wird, Aussagen über Eigenschaften von Programmen in der Sprache zu beweisen (wie etwa die Freiheit von Deadlocks), sind jedoch Beweise für eine konkrete Sprachdefinition nach Überzeugung des Autors dieser Arbeit auf absehbare Zeit nicht praktisch führbar. Nichtsdestotrotz ist es lohnend, eine formale Sprachdefinition zu schaffen, da in diesem Prozess viele unklare Aspekte der informalen Sprachdefinition aufgedeckt und beseitigt werden können. Sofern man ein geeignetes Kalkül wählt (wie das der Abstract State Machines), liefert die formale Sprachdefinition eine Verhaltensaussage für *jedes* Programm der Sprache; es kann also nicht passieren, dass die formale Definition Aspekte des Verhaltens im unklaren läßt.

Auch wenn mit dieser Arbeit eine prototypische Implementierung von SDL-2000 aus der formalen Sprachdefinition abgeleitet wurde, so sollte und kann diese Implementierung jedoch nicht die separat entwickelten Implementierungen von SDL-2000-Werkzeugen ersetzen: Diese Werkzeuge bieten Anpassungen an die Umgebung, in die sich das SDL-Programm einbetten muss und treffen dabei Annahmen etwa über die Zielcomputer und deren Vernetzung; diese Aspekte sind in der formalen Semantikdefinition bewusst nicht berücksichtigt.

10.2 Entwicklungsstand der Sprachdefinition und der Werkzeuge

Mit dieser Arbeit wurde eine *vollständige* formale Definition der Datentypsemantik für SDL-2000 geschaffen. Allerdings wurde bisher nicht für alle Konzepte überprüft, ob die Definition auch *richtig* ist.

Da sowohl die formale Semantik von SDL-2000 als auch der Datentypteil gegenüber früheren Sprachversionen grundlegend geändert wurde, ist zu erwarten, dass auch nach Abschluss dieser Arbeit semantische Probleme sowohl in der informalen wie auch in der formalen Sprachdefinition verbleiben. Nimmt man die während der Entwicklung der Sprachdefinition entdeckten Probleme als Ausgangspunkt, so ist bei der Weiterentwicklung mit Problemen zu rechnen.

Diese Probleme werden sich gleichermassen in der statischen Semantik als auch in der dynamischen Semantik zeigen. Für die statische Semantik sind Probleme insbesondere der folgenden Klassen zu erwarten:

- Die Korrektheitsbedingungen sind vermutlich teilweise falsch: Für die Arbeit wurde auf die Überprüfung der Korrektheitsbedingungen der statischen Semantikanalyse verzichtet. Diese Fehler werden sich zum einen darin äußern, dass Spezifikationen, die eigentlich als richtig gelten sollen, als falsch abgewiesen werden, zum anderen werden Spezifikationen akzeptiert, die eigentlich abgelehnt werden müssten.
- Für manche SDL-Spezifikationen wird die statische Semantikanalyse einen Laufzeitfehler liefern, da Funktionen für Argumente aufgerufen werden, für die sie nicht definiert sind.

Für Spezifikationen, für die die statische Semantikanalyse erfolgreich terminiert, wird die dynamische Definition unter Umständen nicht das erwartete Verhalten zeigen.

In der Definition der formalen Semantik wurden eine Reihe von Annahmen gemacht, deren Wahrheit nicht offensichtlich ist. Diese Annahmen stellen Beweisverpflichtungen dar, für die zur überzeugenden semantischen Fundierung auch Beweise gefunden werden müssen. Dazu gehören die folgenden Annahmen:

- Die Wert der Korrektheitsbedingungen (wahr oder falsch) ist unabhängig davon, ob Transformationen der abstrakten Syntax 0 durchgeführt wurden oder nicht.
- Die statische Analyse (Transformationen, Korrektheitsbedingungen, Mapping-Funktion) terminiert.
- Das Transformationsergebnis hängt nicht von der Reihenfolge ab, in der gleichzeitig gültige Transformationen ausgeführt werden.
- Alle Funktionen werden stets typrichtig verwendet.

Es ist denkbar, dass sich einige dieser Eigenschaften automatisch beweisen lassen; beispielsweise kann man automatisch feststellen, welche Funktionen garantiert terminieren, da sie nicht rekursiv definiert werden und nur über endlichen Mengen operieren. Genauso kann automatisch festgestellt werden, an welchen Stellen möglicherweise Typverletzungen auftreten, weil etwa ein Wert statisch einer Menge M zugeschrieben werden kann, aber als Argument einer Funktion verwendet wird, die nur eine Teilmenge von M verarbeiten kann.

Neben diesen Problemen, die sich aus Unzulänglichkeiten der formalen Semantikdefinition ergeben, können potentiell auch weitere Probleme in den Werkzeugen liegen. In dieser Arbeit wurden die Werkzeuge allerdings so verbessert, dass Unvollkommenheiten in den Werkzeugen, wenn möglich, zu einer Fehlerausgabe führen (wenn etwa Konstrukte verwendet werden, die von den Werkzeugen nicht unterstützt sind). Liefern die Werkzeuge also ein Ergebnis, anstatt abzubrechen, so entspricht dieses Ergebnis in der Regel auch der formalen Sprachdefinition. Sofern die Werkzeuge vorzeitig abbrechen, gibt die Fehlermeldung Auskunft darüber, dass ein Problem in den Werkzeugen entdeckt wurde.

Damit bilden die Werkzeuge einen aus der Sprachdefinition automatisch abgeleiteten Referenzcompiler: Sofern der Compiler erfolgreich zu einem Ergebnis kommt, ist dieses Ergebnis auch dasjenige, welches durch die formale Sprachdefinition vorgegeben wird.

10.3 Verwandte Arbeiten

Verschiedene Arbeiten haben das ASM-Kalkül als Basis einer formalen Sprachdefinition verwendet. So wurde in [GH93] die Semantik von C, in [Wal95] die Semantik von C++, in [Val93] die Semantik von COBOL, in [BS98] und [Wal97] die Semantik von Java, in [BR94] die Semantik von Prolog, in [CH00] die Semantik von Standard ML in [BGM95] die Semantik von VHDL definiert. Diese Arbeiten unterscheiden sich von der SDL-Sprachdefinition auf folgende Weise:

1. In keiner dieser Arbeiten ist die so entstandene Sprachdefinition die „offizielle“ Sprachdefinition. Stattdessen gibt es unter Umständen mehrere konkurrierende formale Sprachdefinitionen, wie beispielsweise für Java. Teilweise konkurrieren diese Definitionen auch mit dem der Formalisierung dieser Sprache durch andere Kalküle. Für SDL steht die formale Sprachdefinition gleichbedeutend mit der informalen Sprachdefinition. Falls Konflikte zwischen beiden Definitionen entdeckt werden, muss das Normierungsgremium diesen Konflikt beseitigen. Allerdings haben die Sprachdefinitionen teilweise trotzdem die Sprachentwicklung beeinflusst, so ist aufgrund der VHDL-Formalisierung ein Defekt in VHDL entdeckt worden, der zur Änderung des VHDL-Standards führen wird [Glä02].
2. Viele Arbeiten decken nur Teile der Sprache ab. So ignorieren beispielsweise die formalen Sprachdefinitionen von C und C++ die Semantik des Präprozessors sowie die statische Semantik (also die Auflösung von Bezeichnern und die andere Korrektheitsbedingungen).
3. Viele Arbeiten setzen keine Werkzeuge für die Verarbeitung der formalen Sprachdefinition oder zur Ausführung von Programmen der Sprache ein, zumindest erwähnen viele Autoren keine derartigen Ergebnisse. Dies liegt in vielen Fällen daran, dass die Autoren sich auf Teile der Sprache beschränken und insbesondere die statische Semantik ignorieren. Damit können sie aus der formalen Sprachdefinition aber keine Werkzeuge mehr ableiten, die beliebige Programme der Sprache akzeptieren und ausführen.
4. Keine dieser Arbeiten definiert die Datentypsemantik über eine funktionale Schnittstelle. Da die meisten der Sprachen keine Unterscheidung zwischen aktiven Objekten und passiven Daten kennen, liegt die Einführung einer funktionalen Schnittstelle für diese Sprachen auch nicht nahe – vielmehr wird die Abarbeitung von Operationen auf Daten direkt in Form von ASM-Programmen ausgedrückt.

Auch wenn nicht alle dieser Beobachtungen auf alle formalen Sprachdefinitionen zutreffen, so ist doch SDL-2000 eine von wenigen Sprachen, für die die offizielle Sprachdefinition eine ausführbare formale Semantik enthält, für deren Ausführung auch tatsächlich Werkzeuge vorhanden sind.

10.4 Ausblick

Mit dieser Arbeit wurde ein Teilergebnis auf dem Gebiet formaler Semantik von Programmiersprachen erzielt. Sie kann zum einen als Ausgangspunkt weiterer Forschung dienen, zum anderen können die Ergebnisse in der Praxis eingesetzt werden.

Mit Hilfe der formalen Semantik von SDL-2000 kann die Normkonformität von SDL-Werkzeugen untersucht werden. Auch wenn sich die entstandenen Werkzeuge selbst nicht als Grundlage eines Softwareproduktionssystems eignen, so können sie doch zur Bewertung solcher Systeme herangezogen werden. Beispielsweise kann man versuchen, das Verhalten eines SDL-Compilers mit dem von SDLC zu vergleichen und exemplarisch festzustellen, ob beide Werkzeuge die gleichen SDL-Programme verarbeiten.

In der Anwendung der Werkzeuge und der formalen Semantik wird man verschiedene weitere Probleme entdecken (siehe Abschnitt 10.2); diese gilt es zu beheben.

Zur Weiterentwicklung formaler Methoden kann man versuchen, auf Basis der formalen SDL-Semantik Aussagen über SDL-Programme zu gewinnen, indem man beispielsweise versucht, Aussagen über ein SDL-Programm zu beweisen. Dabei kann man einerseits versuchen, aus der spezifischen Definition von SDL-2000 Schlüsse zu ziehen, und andererseits diese Semantikdefinition lediglich als Spezialfall eines ASM-Programms zu betrachten und Ergebnisse der ASM-Forschung auf SDL zu übertragen. Aufgrund der Größe der formalen SDL-Definition ist es unwahrscheinlich, dass „manuelle“ Beweise über Programmverhalten gefunden werden können; mehr Erfolg versprechen die Verfahren des *model checking*.

Schließlich sollte man versuchen, die in dieser Arbeit vorgestellten Methoden auf andere Sprachen zu übertragen, um auch für diese Sprachen aus der Reinheit und Klarheit einer formalen Definition Nutzen zu ziehen.

11 Literatur

Arc01 T. Archer. Inside C#. Microsoft Press, 2001.

ASU86 A. V. Aho, R. Sethi, J. D. Ullman. Compilers. Principles, Techniques, and Tools. Addison-Wesley, 1986.

BGM95 E. Börger, U. Glässer, W. Muller. Formal Definition of an Abstract VHDL'93 Simulator By EA-Machines. In C. Delgado Kloos and P.T. Breuer (Hrsg.). Formal Semantics for VHDL. Kluwer Academic Publishers, 1995.

BR94 E. Börger, E. Riccobene. A mathematical definition of full Prolog. In Science of Computer Programming, 1994.

Broy91 M. Broy: Towards a Formal Foundation of the Specification and Description Language SDL. In C. B. Jones, D. J. Cooke, J. M. Wing, Formal Aspects of Computing, Springer, 1991.

BS98 E. Börger, W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java“. In J. Alves-Foss (Hrsg.). Formal Syntax and Semantics of Java. Springer Lecture Notes in Computer Science, Springer, 1998.

CH00 S. C. Cater, K. Huggins, An ASM Dynamic Semantics for Standard ML. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (Hrsg.), Abstract State Machines -- ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings, TIK-Report 87, Eidgenössische Technische Hochschule Zürich, 2000.

EB93 P. van Eijk, A. Belifante: The Term Processor Kimwitu. University of Twente, 1993.

EGG+01 R. Eschbach, U. Glässer, R. Gotzhein, M. v. Löwis, A. Prinz: Formal Definition of SDL-2000 – Compiling and Running SDL Specifications as ASM Models. In E. Börger, U. Glässer, H. Maurer, C. Calude, A. Salomaa, K. Tochtermann, Journal of Universal Computer Science, Volume 7, No. 11, Springer, 2001.

Dol01 L. Doldi. SDL Illustrated - Visually design executable models. Privat veröffentlicht von L. Doldi, Toulouse, 2001.

FKSH72 H. Frühauf, W. Kämmerer, K. Schröder, H. Thiele, H. Völz (Herausgeber): Bericht über die algorithmische Sprache ALGOL 68. Elektronisches Rechnen und Regeln, Sonderband 15, Akademie-Verlag, Berlin, 1972.

FLP95 J. Fischer, S. Lau, A. Prinz: A Short Note About BSDL – Semantic Issues for SDL. SDL-Newsletter 18, 1995.

FSF98 Free Software Foundation. Flex – a scanner generator. http://www.gnu.org/manual/flex/, 1998.

FSF02a Free Software Foundation. Bison 1.35. http://www.gnu.org/manual/bison/, 2002.

FSF02b Free Software Foundation. Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/, 2002.

GGRS98 R. Gotzhein, B. Geppert, F. Rößler, P. Schaible: Towards a New Formal SDL Semantics. In Y. Lahav, A. Wolisz, J. Fischer, E. Holz (Hrsg.), Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC, SAM98, Berlin, 1998.

GH93 Y. Gurevich, J. K. Huggins. The Semantics of the C Programming Language. Springer Lecture Notes in Computer Science 702, Springer, 1993.

GJSB00 J. Gosling, B. Joy, G. Steele, G. Bracha. The Java Language Specification. 2. Auflage, Addison-Wesley, 2000.

Glä02 U. Glässer. Abstract State Machines: Programming Languages. http://www.uni-paderborn.de/cs/asm/Available_Materials/proglang.html, 2002.

Gur95 Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, Specification and Validation Methods. Oxford University Press, 1995.

Gur97 Y. Gurevich. ASM Guide 97. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan-Ann Arbor, 1997.

ISO646 ISO. Information technology -- ISO 7-bit coded character set for information interchange. ISO/IEC 646:1991.

ISO2022 ISO. Information technology -- Character code structure and extension techniques. ISO/IEC 2022:1994.

ISO8485 ISO. Programming languages -- APL. ISO/IEC 8485:1989.

ISO8807 ISO. Information processing systems -- Open Systems Interconnection -- LOTOS -- A formal description technique based on the temporal ordering of observational behaviour. ISO/IEC 8807:1989.

ISO9899 ISO. Programming languages -- C. ISO/IEC 9899:1999.

ISO10646 ISO. Information Technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. ISO/IEC 10646:2000.

ISO14882 ISO. Programming languages -- C++. ISO/IEC 14882:1998.

KCR98 R. Kelsey, W. Clinger, J. Rees (Herausgeber). Revised^5 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices, Vol. 33, No. 9, October, 1998.

Lis87 B. Liskov. Keynote address – data abstraction and hierarchy. ACM SIGPLAN Notices, Addendum to the proceedings on Object-oriented programming systems, languages and applications, Volume 23 Issue 5, 1987.

vL97 v.Löwis, M.: Using CORBA in an SDL Simulation Environment. DOCT Frankfurt, 1997.

vLF00 M. v. Löwis, N. Fischbeck. Python 2. Addison-Wesley, Bonn, 2000.

LMB92 J. Levine, T. Mason, D. Brown. lex & yacc. 2. Auflage, O'Reilly, 1992.

LP95 S. Lau and A. Prinz. BSDL: The Language – Version 0.2. Institut für Informatik, Humboldt-Universität zu Berlin, 1995.

vLP02 M. v. Löwis, M. Piefel. The Term Processor Kimwitu++. Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics, Orlando, 2002.

Mat01 Mathematik-Olympiade-Siegerland e.V. Mathe-Lexikon. http://san-pc.hrz.uni-siegen.de/olympia/Lexikon

McC79 J. McCarthy. History of Lisp. Artificial Intelligence Laboratory, Stanford University, 1979.

McG02 McGraw-Hill. Encyclopedia of Science & Technology. 9th Edition, McGraw-Hill, 2002.

ML86 M. Marcotty, H. Ledgard. The World of Programming Languages, Springer-Verlag, Berlin 1986.

Mic99 Microsoft Corporation. Microsoft® Office 2000 Resource Kit. Microsoft Press, 1999.

Mic02 Microsoft Research. AsmL for Microsoft .NET. Microsoft Corporation, 2002.

OFM+94 A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, J. R. W. Smith. Systems Engineering Using SDL-92. North-Holland, 1994.

OMG01 Object Management Group. OMG Unified Modelling Language Specification. Version 1.4, formal/01-09-67, Framingham, 2001.

OMG02a Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.6.1, formal/02-05-8, Framingham, 2002.

OMG02b Object Management Group. Naming Service. Version 1.2, formal/02-09-02, Framingham, 2002.

Pie00 M. Piefel. Ein automatisch generierter SDL-Compiler. Diplomarbeit, Humboldt-Universität zu Berlin, 2000.

Pie02 M. Piefel. Der Termprozessor Kimwitu++. http://site.informatik.hu-berlin.de/kimwitu++/, 2002.

Pri99 A. Prinz. Formal Semantics for SDL - Definition and Implementation. Habilitation, Humboldt-Universität zu Berlin, 2001.

PS58 A. J. Perlis, K. Samelson: Preliminary Report – International Algebraic Language. Communications of the ACM 1(12), 1958.

Pyt02 PythonLabs. Python Language Home Page, http://www.python.org/, 2002.

Q.1248 ITU. Interface recommendation for Intelligent Network Capability Set 4. Recommendation Q.1248, Genf, 2001.

Sch94 R. Schröder: SDL‘92 data handling in combination with ASN.1. master thesis, Department of Computer Science, Humboldt University Berlin, Germany, March 1994.

Sch02 R. Schröder. SDL-Datenkonzepte – Analyse und Verbesserungen, Dissertation, Humboldt-Universität zu Berlin, 2002.

SITE Project SITE: http://www.informatik.hu-berlin.de/Themen/SITE.

Sto77 J. E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, 1977.

Tar77 A. Tarski. Einführung in die mathematische Logik. Vandenhoeck & Ruprecht, 1977.

Tur36 A. Turing, On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Series 2, 1936.

Val93 M. Vale. The Evolving Algebra Semantics of COBOL. Part 1: Programs and Control. University of Michigan EECS Department Technical Report CSE-TR-162-93.

Wal95 C. Wallace. The Semantics of the C++ Programming Language. In E. Börger (Hrsg.), Specification and Validation Methods. Oxford University Press, 1995.

Wal97 C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. University of Michigan EECS Department Technical Report CSE-TR-355-97

X.680 ITU. Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. Recommendation X.680, Genf, 1997.

Z.100-92 ITU. CCITT Specification and Description Language, Recommendation Z.100, Genf, 1993.

Z.100-00 ITU. ITU-T Specification and Description Language, Recommendation Z.100, Genf, 1999.

Z.100-01 ITU. ITU-T Specification and Description Language, Recommendation Z.100 Ammendment 1, Genf, 2001.

Z.100F ITU. ITU-T Specification and Description Language – Formal Semantics, Recommendation Z.100 Annex F, Genf, 2000.

Z.105 CCITT: SDL in combination with ASN.1, International Standard Recommendation Z.105, Genf, 2000

Z.130 ITU: ITU-ODL - ITU Object Description Language, Recommendation Z.130, Genf, 1999.

A Index

Erklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift „Formale Semantik des Datentypmodells von SDL-2000“ selbständig und ohne unerlaubt Hilfe angefertigt habe;
- ich mich nicht bereits anderwärtig um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Tabellarischer Lebenslauf

Addresse	Wassermannstr. 71 12489 Berlin
Geburtsdatum	25. April 1970
Staatsangehörigkeit	Deutsch
Vater	Michael v. Löwis of Menar
Mutter	Gerlind v. Löwis of Menar, geb. Merkel
1976-1986	2. POS Köpenick
1987-1988	Abitur an der Spezialklasse für Mathematik und Physik der Humboldt-Universität zu Berlin
1988-1990	Dienst bei der NVA
1990-1993	Informatik-Studium an der Humboldt-Universität; Abschluss als Diplom-Informatiker
1994	Studium an der California State University, Fresno Abschluss als Master of Science in Computer Science
1995-2002	Wissenschaftlicher Mitarbeiter an der Humboldt-Universität
ab 2003	Wissenschaftlicher Mitarbeiter am Hasso-Plattner-Institut in Potsdam